

A WORKSHOP ON THE GATHERING OF INFORMATION
FOR PROBLEM FORMULATION

Albert N. Badre
Georgia Tech Research Institute

AD A127507

BASIC RESEARCH



U. S. Army

Research Institute for the Behavioral and Social Sciences

September 1981

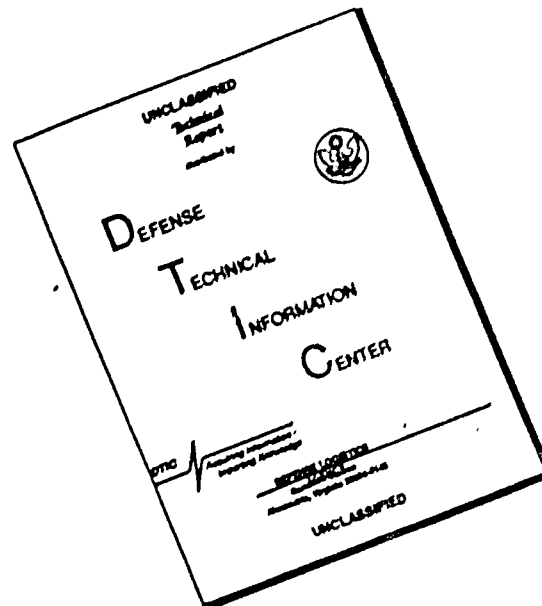
Approved for public release; distribution unlimited.

DTIC
ELECTE
APR 29 1983
S D E

DTIC FILE COPY

83 04 28 118

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER Research Note 83-9	2. GOVT ACCESSION NO. AD-A127587	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A WORKSHOP ON THE GATHERING OF INFORMATION FOR PROBLEM FORMULATION.		5. TYPE OF REPORT & PERIOD COVERED Final Report 3/1/80 - 9/1/81
		6. PERFORMING ORG. REPORT NUMBER G36-651
7. AUTHOR(s) Albert N. Badre		8. CONTRACT OR GRANT NUMBER(s) MDA 903-80-C-0144 and Modification No. 1
9. PERFORMING ORGANIZATION NAME AND ADDRESS Georgia Tech Research Institute School of Information and Computer Science Atlanta, Georgia 30332		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 2Q161102B74F
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Supply Service - Washington Room 10-245, The Pentagon Washington, D.C. 20310		12. REPORT DATE September 1, 1981
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office) Office of Naval Research Resident Representative 325 Hinman Research Building Ga. Institute of Technology, Atlanta, GA 30332		13. NUMBER OF PAGES 148
		15. SECURITY CLASS. (of this report) Unclassified
16. DISTRIBUTION STATEMENT (of this Report) Approved for Public Release; Distribution Unlimited		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Human-Computer Interface Interactivity Information Processing		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The purpose of this workshop was to bring together a group of research scientists from various disciplines to discuss and report their research findings on the topic of problem representation for interactive information processing. During the planning phases of the project, it was agreed that		

the proposed general topic should be limited to the problems of representation and information processing in the context of human-computer interface. Based on this theme, a set of topics were developed and used to select and organize speakers and panels. These were:

1. Psycholinguistic factors in computer communication;
2. Compatible knowledge and memory structures for computer communication;
3. Representing and structuring displayed information in computer communication; *in 1*
4. Representing information for decision, learning, and help processes in computer communication.

The end result was a very successful workshop that included a total of twenty presentations and forty participants.

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	



Table of Contents

	<u>PAGE</u>
Table of Contents.....	i
Schedule of Events.....	1
Albert N. Badre.....	2
Introduction	
Richard Burton.....	4
Experiences with a Natural Language Interface to an ICAI System	
Jaime G. Carbonell.....	5
Towards a Robust, Task-Oriented Natural Language Interface	
Sam L. Ehrenreich.....	14
Creating an Algorithm for Generating Abbreviations to be Used in User-Computer Transactions	
Jim Foley.....	16
Tools for Designer of User Interface	
George W. Furnas.....	28
Psychological Structure in Information Organization and Retrieval: Arguments for More Considered Approaches, and Work in Progress	
Mark D. Jackson and Judith E. Tschirgi.....	32
The Nature of User-Generated Commands for Interacting with a Computer	

Janet Kolodner.....	34
A Conceptual Approach to Natural Language Fact Retrieval	
Thomas K. Landauer and Susan T. Dumais.....	45
Psychological Investigations of Natural Command and Query Terminology	
Michael Lebowitz.....	48
Organizing Memory for Use in Understanding	
Mark Miller and Paul R. Michaelis.....	55
Artificial Intelligence and Human Factors: A Necessary Synergism for the Interface of the Future	
Franklin L. Moses.....	87
Overview of Selected Display Formatting and Clutter Reduction Techniques	
Phyllis Reisner.....	92
Formal Grammar Representation of Man-Machine Interaction	
Elaine Rich and Aaron Temin.....	96
A Role Based Help System for Scribe	
Michael L. Schneider.....	107
Models for the Design of Static, Software Systems	
Ben Shneiderman.....	118
System Message Guidelines: Positive Tone, Constructive, Specific, and User Centered	
Elliot Soloway and Jeff Bonar.....	125
Empirical Evaluation with Novice Users of Some Programming Language Constructs	

Albert L. Stevens, Michael D. Williams, and James D. Hollan.....134

An Advanced Human Interface for Computer Assisted
Instruction in Propulsion Engineering

John C. Thomas.....135

Metamorphosis through Metaphor

Michael D. Williams and J. Hollan.....140

A System for Computer Aided Memorization

APPENDIX A.....142
Names and Addresses of Participants

Workshop on Human Computer Interaction

Revised Schedule

Thursday, March 26, 1981

- 9:30 - 10:00 Coffee and Doughnuts
- 10:00 - 10:45 Opening Session
A. Badre
S. Halpin
- 10:45 - 12:30 Modeling the User
E. Rich and A. Temin
M. Schneider
E. Soloway and J. Bonar
- 12:30 - 1:30 Luncheon
- 1:30 - 3:00 Interfaces - Development
J. Foley
M. Miller and P. Michaelis
J. Thomas
- 3:00 - 3:30 Break - Coffee and Softdrinks
- 3:30 - 5:00 Designing Intelligent Interfaces
R. Burton
J. Carbonell
A. Stevens, M. Williams, and J. Hollan

Friday, March 27, 1981

- 8:30 - 9:00 Coffee and Doughnuts
- 9:00 - 11:00 Human Factors of Interactive Languages
S. Ehrenreich
G. Furnas
T. Landauer and S. Dumais
M. Jackson and J. Tschirgi
- 11:00 - 11:15 Break
- 11:15 - 12:45 Memory Structures for Human-Computer Communication
J. Kolodner
M. Lebowitz
M. Williams and J. Hollan
- 12:45 - 1:45 Luncheon
- 1:45 - 3:00 Messages and Displays
F. Moses
P. Reisner
B. Shneiderman
- 3:00 - 3:15 Break - Coffee and Soft Drinks
- 3:15 - 4:15 General Discussion and Summary
- 4:30 - 6:00 Panel

THE HUMAN COMPUTER INTERFACE

INTRODUCTORY REMARKS

Albert N. Badre

When asked to sit down at a computer terminal and perform what is considered an elementary task, most novice operators are likely to be confused and frustrated. Even the simplest of tasks seems to require an excessive level of computer sophistication or the motivation to read and understand an over abundance of accompanying documentation.

The population of computer users is growing at a very rapid pace, and an increasingly large number of this generation of new users is not data processing or computer trained. Yet,

- the language that the operator must use to interact with the machine
- the documentation, whether on-line or off-line, that he/she has to read in order to learn how to instruct the machine; and
- the system messages that are displayed

are couched in the vocabulary and language habits of the computer expert.

Accordingly there is a growing consensus in the computer science community that the user-compatibility of the human interface should be considered and incorporated into the design of all computer systems at the initial stages of development. "Information processing" systems are likely to be more user compatible if they are designed to adapt to the information processing capabilities and limitations of the user. It is becoming, therefore, increasingly necessary to explore and identify the human information processing factors, constraints, and variables that are associated with making the interface more user compatible. This means identifying and considering factors relating to what the operator "does" at the display station in order to perform a desired task and what the system does in return.

In this workshop symposium we will be dealing with six inter-related topics that revolve around the user interface theme. These are: Modeling the user, interface development factors, design considerations for intelligent and adaptive interfaces, memory structures, the human factors of language interaction, and messages and displays.

Experiences with a Natural Language
Interface to an ICAI System

Richard Burton

Towards a Robust, Task-Oriented Natural Language Interface

Jaime G. Carbonell
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

This paper analyzes the inception of a new generation of robust, task-oriented natural language interfaces in light of new theoretical advances and analysis to avoid limitations of previous efforts. Three key ideas are discussed: 1) dynamic selection of parsing strategies, 2) exploiting domain-specific semantics and grammatical constructions, and 3) integrating recent theoretical findings into task-oriented parsing. An implemented natural language interface conforming to some of the new objectives is discussed, as are current plans for a more-general-scope natural language interface.

3 February 1981

Towards a Robust, Task-Oriented Natural Language Interface

Jaime G. Carbonell
Carnegie-Mellon University
Pittsburgh, PA 15213

1. Objectives and Historical Perspective

Natural language comprehension has been studied from two primary perspectives in Artificial Intelligence:

- As a vehicle to investigate and simulate human cognitive processes embodying components of either a linguistic or psychological theory of language comprehension.
- As a means of implementing task-oriented "natural language front ends" to complex computer systems.

The "basic science" approach has produced some significant principles and techniques (e.g., expectation-based language analyzers [7, 1]), but no truly robust parsers for computer-naive users have been developed in this paradigm.

The applied "engineering" approach has proceeded by either building the domain of application into the parser itself, or by relying on syntax-only linguistic parsers. Neither approach has proven wholly satisfactory. The former suffers from virtual lack of transferability to new domains, while the latter suffers from extreme fragility: the inability to cope with any input not strictly conforming with its rigid internal grammar. However, it must be noted that some successful parsers have emerged from these limited approaches, such as LIFER [5] and LUNAR [8]. Both of these efforts, unfortunately, required man-years of development and tuning before their performance approached the user-acceptance level. Their primary contributions were in the computational mechanisms they introduced, which could later be incorporated into more sophisticated parsers.

A major objective in the design of task-oriented parsers is to provide the user maximal flexibility (within the semantics of the domain) to express his utterance. For example, the graceful interaction project [4] is a recent attempt at coping with limited ungrammaticality in a task-oriented parser. The means by which recent task-oriented parsers strive for robustness and flexibility is to incorporate domain semantics into their parsing knowledge bases (but not into the programs themselves). Here, we go one step further and exploit domain knowledge to dynamically choose the optimal parsing strategies. Moreover, the work described in this paper attempts to take full advantage of lessons learned from more theoretical natural language research. Our objectives can be summarized as follows:

- Create a robust parser, in the sense that it must tolerate common ungrammaticality, ellipsed constructions, and different phrasings within its domain of application.
- Implement the parser in a modular manner with respect to its knowledge sources. This means that domain knowledge necessary for the parser ought to be divorced from the program, from general semantic knowledge, and from linguistic knowledge. Hence, only one knowledge base need be altered in transferring the parser to a new application domain. The program itself is general with respect to the choice of task domain.
- Exploit new advances in natural language processing not previously incorporated into task-oriented parsers. Some well-established powerful methods developed to simulate human language understanding (most notably expectation-based disambiguation) have not previously been used in task-oriented approaches, although they have proven computationally effective in more general domains.
- Minimize the time required to transfer the parser to a new domain. This goal is furthered by our modularity consideration, but in addition I want to work towards a uniform method of incorporating new domain knowledge, including knowledge of technical jargon particular to a given domain.

In order to further these ends I developed an initial parser that combines partial pattern matching, semantic-grammars [5] and equivalence transformations. I applied this parser to the task of building and querying a semantic-network [2] data base. The central lesson learned from this exercise is that the combination of the three parsing strategies yields not only a more robust parser than a single-strategy method, but surprisingly the time it took develop its domain application (admittedly not a very complex task) was considerably less than expected (less than three weeks).

A crucial (and perhaps unintuitive) fallacy of previous task-oriented parsers is their commitment to a simple uniform parsing strategy. Since natural language is a complex phenomenon (even in task-oriented domains), this design criterion had the effect of pushing the complexities into the domain grammars, dictionaries and other domain-specific components of the parser. In the clearer vision of hindsight, this design decision greatly complicated the application of existing parsers to new domains. Is it not more desirable to incorporate all the decision-making complexities required to parse natural language structures into the kernel program itself? Once built, this program need not be redesigned for a new task domain. Minimizing the requisite complexity and size of domain-dependent components is an extremely productive venture. Parsing-strategy selection, semantic matching routines, and other domain-independent components should be provided as a *kernel parser*, which is augmented by domain-specific knowledge bases in each applications domain.

In designing the kernel parser, a dominant criterion is that it select the parsing strategy in accordance with the type of natural language construct it attempts to parse. Some information can be expressed more naturally and more parsimoniously in one form (e.g., linear patterns) while other information is best expressed as case structures, equivalence transformations, or semantic grammar

productions. To illustrate this point, I attempted to encode all the knowledge in my parser as a pure semantic grammar. This task has more than tripled the size of the task-specific knowledge base, and I have not yet finished (nor do I intend to finish) the conversion. The primary reason for the relative increase in size is that much of the information must be stated with a high degree of redundancy and often in an awkward, round-about manner when it must be coerced into a uniform, context-free representation.

2. The DYPAR Parser

DYPAR¹ combines three parsing strategies:

- **A context-free semantic grammar component**, grouping domain information into hierarchical semantic categories useful in classifying individual words and phrases in the input language.
- **A partial pattern match component**, represented as pattern-action rules. The patterns may contain individual words, semantic categories (from the semantic grammar), wild cards, optional constituents, register assignment and register reference. This method enables the semantic grammar non-terminal categories to be applied in a much more effective context-sensitive manner than would be the case is a pure context-free grammar recognizer.
- **Equivalence transformations** map domain-dependent and domain-independent constructs into canonical form, requiring a fraction of the patterns and semantic categories that would otherwise be necessitated. If a phrase-structure can be expressed in several different ways, while retaining the same meaning, it is clearly beneficial to first map it into canonical form, rather than being forced to include all possible variants in every context where that constituent could occur.

Below I give an example of each type of linguistic information used in DYPAR. In order to understand these examples, a few notational conventions must be introduced: <BRACKETS> denote a non-terminal semantic grammar symbol. A word starting with an exclamation mark (e.g., !REGISTER) denotes the name of register. A vertical bar (|) denotes disjunction in a pattern. A # in a pattern matches a single word. An asterisk (*) matches an arbitrary sequence of words. The construction (!REGISTER pattern) assigns whatever matches the pattern to the register specified. A colon (:) before a constituent in a pattern indicates that constituent is optional.

DYPAR, as we see in the dialog below, is the front end of a semantic network data-base update and query system. Therefore, its domain knowledge consists of language constructs relevant to this task. First, consider a fragment of its semantic grammar:

¹Robust multi-strategy "DyNamic PARsing" is still in its infant stages, requiring frequent changes.

```

<INFO-REQ> -> (<WHAT-Q> | <INFO-REQ1>]
<INFO-REQ1> -> (: <POLITE> <INFO-REQ2> : <WHAT-Q>]
<INFO-REQ2> -> (TELL <me-US> . ABOUT | GIVE <me-US> | PRINT | TYPE ]

```

This fragment, together with the rewrite rules for the other non-terminals above (e.g., <BE-PRES>, whose rewrite is all the present-tense conjugations of the verb "to be") recognizes the initial segment of information-request queries such as: "What is ...", "Tell me what is ...", "Tell me about...", "Would you give me ...", etc.

Now, consider a pattern-match rule:

```

(: <det> (!val #) <be-pres> : <DET> (!PROP #) OF
   : <DET> (!NAM #) : <dpunct>))
=>
(LTM-STORE !NAM !VAL !PROP)

```

This rule recognizes sentences such as: "Felix is a friend of Fido", or "Reagan is president of the USA", and passes the information to the data base manager for consistency checking and storage. In order to pass the information gathered in the pattern match process, the registers are assigned appropriate values. For instance, in the second example, !NAM is assigned "USA", !PROP is assigned "president" and !VAL is assigned "Reagan".

The equivalence transformations also use the pattern matcher. For instance, consider the following simple (but useful) transformation:

```

((!S1 *) (!W1 #) <POSS> (!W2 #) (!S2 *) : (!P <PUNCT>))
::>
(NCONC !S1 !W2 (LIST 'OF) !W1 !S2 !P)

```

This transformation maps possessive constructions into attribute-value constructions, which we chose as canonical. For instance "Tell me about the VAX-785's performance." is mapped into "Tell me about the performance of the VAX-785." The latter construction is recognized by a pattern-action rule. Since possessive constructions can occur in many contexts, the single transformation above saves us from duplicating pattern match rules for each different context where an attribute-value construction can occur.

The control structure of DYPAR is essentially an applicative condition-action cycle, which halts upon no rules being applicable (in which case a partial pattern-match strategy is attempted), or upon one of the pattern-action rules firing and passing its information to the data-base manager. Let us see a brief dialog with DYPAR. For simplicity, we start out with an empty data base. Items in *italics* below were typed in by the user. Everything else was output by the system. Items in UPPER CASE are paraphrases of internal semantic network relations output by a rudimentary English generator.

**(parser)*

You can build and query a simple semantic network

by stating information and questions in English.

+Fido is a nice dog.

Storing assertion in semantic net: FIDO IS DOG.

[COMMENT: When DYPAR does not recognize concepts in the input, it prompts the user for additional information required by the integrity-checker in the data-base manager.]

What is NICE?

+Nice is a disposition.

Storing assertion in semantic net: NICE IS DISPOSITION.

Is it correct to say that THE DISPOSITION OF FIDO IS NICE?

(Y,N): Yes.

Adding new assertion: THE DISPOSITION OF FIDO IS NICE.

+Fido's cousin is Felix.

Transforming (FIDO \$S COUSIN IS FELIX PERIOD)

Into ::> (THE COUSIN OF FIDO IS FELIX PERIOD)

Transforming (THE COUSIN OF FIDO IS FELIX PERIOD)

Into ::> (FELIX IS THE COUSIN OF FIDO PERIOD)

Adding new assertion: THE COUSIN OF FIDO IS FELIX.

[COMMENT: To illustrate the parsing process in a bit more detail, the rule tracer was activated for the following interaction.]

+Cousin is a symmetric relation.

Parse is:

Rule RUL19

Action will be: (PROGN (LTM-STORE !NAM !VAL (QUOTE ISA:))
(LTM-SPEC !NAM !VORP NIL NIL T))

With bindings:

(!VAL RELATION)

(!VORP SYMMETRIC)

(!NAM COUSIN)

I already knew that: COUSIN WAS A RELATION.

Is it correct to say that THE RELATION OF COUSIN IS SYMMETRIC?

(Y,N): yes.

Adding new assertion. THE RELATION OF COUSIN IS SYMMETRIC.

Inference: Adding new assertion: THE INVERSE OF COUSIN IS COUSIN.

Inference: Adding new assertion: THE COUSIN OF FELIX IS FIDO.

+Cat chasing is a dog's hobby.

Transforming (CAT-CHASING IS A DOG \$S HOBBY PERIOD)

Into ::> (CAT-CHASING IS A HOBBY OF DOG PERIOD)

Adding new assertion: THE HOBBY OF DOG IS CAT-CHASING.

[COMMENT: Since the program never heard of "dog" or "fido" it does not know whether to express them with or without a determiner. "Fido" is a proper name, therefore it needs no determiner, but "dog" in all the examples above looks strange without one. Now, we fix that problem by providing the program with the necessary information (in English, of course).]

+The concept of dog is generic.

Transforming (THE CONCEPT OF DOG IS GENERIC PERIOD)

Into ::> (GENERIC IS THE NODE-TYPE OF DOG)

[COMMENT: In our domain "concept" and "node-type" are synonyms. In future output, "dog" will be printed with a determiner when appropriate.]

Adding new assertion: THE NODE-TYPE OF DOG IS GENERIC.

+Could you tell me what a dog's hobby is?

Transforming (COULD YOU TELL ME WHAT A DOG \$S HOBBY IS QMARK)

Into ::> (COULD YOU TELL ME WHAT IS A DOG \$S HOBBY QMARK)

Transforming (COULD YOU TELL ME WHAT IS A DOG \$S HOBBY QMARK)

Into ::> (COULD YOU TELL ME WHAT IS THE HOBBY OF A DOG QMARK)

THE HOBBY OF A DOG IS CAT-CHASING.

+give me everything you know about fido

FIDO IS A DOG.

THE HOBBY OF FIDO IS CAT-CHASING.

THE COUSIN OF FIDO IS FFLIX.

THE DISPOSITION OF FIDO IS NICE.

+Napping is the hobby of Fido.

That contradicts what I could infer by inheritance.

THE HOBBY OF FIDO WAS CAT-CHASING.

Should I add the assertion anyway? (Y,N): no.

OK, discarding new assertion.

+Exit this program.

Leaving natural language interface. Back to LISP.

(CPU-SECONDS: 12.056 GC-TIME: 6.780)

As we see in the above example, robust communication with the user requires not only a flexib

domain-oriented parser, but also an interactive query capability and a natural language generator. However, the latter two processes are conceptually simpler, and not the topic of this paper.

3. Future Directions

DYPAR illustrates the harmonious integration of three parsing strategies. However, it is only the first step in exploiting the multi-strategy approach to develop real-world, robust, natural language interfaces. In terms of sophistication, DYPAR straddles the boundary between an advanced toy and a rudimentary real-applications system. One direction of continued development is to enhance the pattern matcher, build additional general transformations, and create a sub-interface to facilitate extensions to the grammar by a domain expert (not necessarily a natural-language expert). A first step in the direction of automating and simplifying user extensibility has been taken in the development of the KLAUS system [6]. At CMU, we are focusing on a complementary, and perhaps more fundamental research direction.

If the gestalt performance of integrating three parsing strategies has proven more effective than the application of any single strategy, why not extrapolate this result to include additional parsing strategies? Indeed, we have designed a flexible control structure for integrating case-instantiation as the central parsing strategy -- calling upon other strategies discussed in this paper, in addition to more domain-specific strategies, when appropriate [3]. Case-frame instantiation is the most general parsing strategy capable of exploiting domain semantics. Hence, it should provide a quantum jump in the general applicability of our task-oriented parser. Moreover, techniques such as expectation-driven disambiguation [7, 1] developed by the non-applied school of natural language processing, can now be brought to bear in real-world applications. The reason why case-frame parsers have not been developed in task-oriented domains is that while they capture general principles admirably, they fail to recognize specific idioms, compound nouns and the like. However, the addition of partial pattern matching (ideally suited to detect idiomatic expressions) integrated with case-frame instantiation and other parsing methods should provide a high degree of generality without sacrificing robustness.

Graceful interaction with the user is a worthy goal for any natural language front end whose users may be computer-naive. People invariably produce ungrammatical utterances, leave out words, add interjections, and use terms outside the vocabulary of any system [4]. It is essential that a real-world system "fail soft" in such circumstances, and interact with the user to enable graceful recovery. We saw some simple examples of this in DYPAR. However, the expectation-setting provided by a case system incorporating domain knowledge can be a more powerful tool to minimize failure.

Consider, for instance, a file-management system where a user may type "Transfer the files in my directory to the accounts directory." It is fairly clear to us humans that the user meant to type "files", even if we know perfectly well that "flies" is a legitimate word in our vocabulary. A case-frame system

knows that the objective case in the transfer imperative (as applied to the file-management domain) requires a logical data entity, which "flies" is not. Realizing this violated semantic requirement, it can proceed to see whether by spelling correction, morphological decomposition, or detecting potential omissions it can map "flies" into a known filler of that case. Here, spelling correction works, and the system can proceed to inform the user of its correction (allowing the user to override if need be).

I conclude by reiterating my central theme: *Integration of multiple parsing strategies is perhaps the single most powerful principle in the development of robust, task-oriented natural language interfaces.*

4. References

1. Birnbaum, L. and Selfridge, M., "Conceptual Analysis in Natural Language," in *Inside Computer Understanding*, R. Schank and C. Riesbeck, eds., New Jersey: Erlbaum Assoc., 1980, pp. 318-353.
2. Brachman, R. J., "On the Epistemological Status of Semantic Networks," in *Associative Networks*, N. V. Findler, ed., New York: Academic Press, 1979.
3. Carbonell, J. G. and Hayes, P. J., "Dynamic Strategy Selection in Flexible Parsing," *Proceedings of the 19th Meeting of the Association for Computational Linguistics*, (Submitted 1981).
4. Hayes, P. J. and Mouradian, G. V., "Flexible Parsing," *Proceedings of the 18th Meeting of the Association for Computational Linguistics*, 1980, pp. 97-103.
5. Hendrix, G. G., Sacerdoti, E. D. and Slocum, J., "Developing a Natural Language Interface to Complex Data," Tech. report Artificial Intelligence Center., SRI International, 1976.
6. Hendrix, G. G. and Haas, N., "Acquiring Knowledge for Information Management," in *Machine Learning*, Michalski, R., Carbonell, J. G. and Mitchell, T., eds., Palo Alto, CA: Tioga Pub. Co., 1981.
7. Riesbeck, C. and Schank, R. C., "Comprehension by Computer: Expectation-Based Analysis of Sentences in Context," Tech. report 78, Computer Science Department, Yale University, 1976.
8. Woods, W., Kaplan, R. and Nash-Webber, B., "The Lunar Sciences Natural Language Information System: Final Report," Tech. report 2378, Bolt Beranek and Newman Report, 1972.

CREATING AN ALGORITHM FOR
GENERATING ABBREVIATIONS TO BE USED
IN USER-COMPUTER TRANSACTIONS

Sam Ehrenreich
US Army Research Institute for the
Behavioral and Social Sciences

The US Army is in the process of developing automated tactical systems. These systems will incorporate a dialogue mode (e.g., form-filling, menu, query language) for communicating between the user and the computer. For the convenience of both, much of this communication will involve abbreviations. The Army Research Institute (ARI) is engaged in preparing an algorithm for use by system designers in creating easy to use abbreviations for these systems. The algorithm will not only be concerned with generating abbreviations for command terms. Rather, the primary domain of the algorithm will be the lexical terms used in exchanging information between the user and the computer.

This summary describes the empirical issues that were investigated in ARI's abbreviation project. The data that was collected, along with an algorithm for generating abbreviations, will be presented at the workshop.

All of the experiments for this project have already been completed. However, a few still remain to be analyzed. The participants used in these experiments were enlisted Army personnel. The stimuli used were words which are likely candidates for abbreviation on an automated tactical system. However, it is believed that the nature of both the participants and the stimuli are such that the resulting algorithm will be applicable for use with most classes of operators and with most sets of words.

The general abbreviation techniques which were considered as candidates for forming the basis of the algorithm are: (1) truncation, i.e., delete all but the first few letters of a word; (2) contraction, i.e., remove all of the word's vowels except for vowels occurring as the first letter; and (3) abbreviation

by the consensus of a committee. In order to create the desired algorithm, the empirical questions which were investigated are:

1. What are people's personal preferences with regard to the abbreviations formed by the different abbreviation techniques?
2. How do the different abbreviation techniques compare when participants are presented with a word and asked to recall its abbreviation (i.e., encoding)? How do the methods compare when the task is decoding?
3. When participants are asked to produce abbreviations of their own choosing, what abbreviation method do they tend to naturally use?
4. When participants' experiences with a word and its abbreviation increases, do the absolute and relative effectiveness of the different abbreviation techniques change?
5. When participants are instructed in the rule system underlying the different abbreviation techniques, do the absolute and relative effectiveness of the abbreviations change?
6. Should abbreviations be of a fixed or variable length?
7. How can different words that result in identical abbreviations be handled (e.g., when using the truncation method, both TRANSLATOR and TRANSPORT are abbreviated as TRAN)?
8. Can endings (e.g., -ed, -ing) be effectively incorporated into abbreviations?

The answers to these questions will represent the empirical basis on which an abbreviation algorithm is formed. The desired algorithm is one which is completely deterministic in the abbreviations it forms. Using the algorithm, the system designer should have minimum input in determining the abbreviation to be created. Although the algorithm that will be created will not be based on a complete investigation of all possible variables, it is expected that it will result in abbreviations which are significantly easier to use than the arbitrary and inconsistent abbreviations presently used on Army systems.

Tools for the Designers of User Interfaces*

James D. Foley

March, 1981

Institute for Information Science and Technology
Department of Electrical Engineering and Computer Science
School of Engineering and Applied Science
The George Washington University

Washington, D.C. 20052

REPORT GWU-IIST-81-07

This paper was presented at the Workshop/Symposium on Human Computer Interaction, sponsored by the U.S. Army Research Institute and Georgia Institute of Technology.

*This work is being carried out by the author and M.B. Feldman, co-principal investigator, H. Holmes, Visiting Scientist from Lawrence Berkeley Laboratory, J. Thomas, Visiting Scientist from Battelle Northwest Laboratories, Research Assistants T. Bleser and G. Rogers, Graduate Research Assistant A. Kamran, and P. Chan. The work is partially sponsored by the U.S. Department of Energy (Grant DE-AS05-79ER1052) and the U.S. Army Research Institute (Grant MDA 903-79-G-01). V.L. Wallace of the University of Kansas is co-principal investigator with the author for the work entitled "Evaluation of Interaction Techniques."

Tools For the Designers of User Interfaces

Our research objective is to develop methodologies and tools which can aid in the design of user-computer interfaces. We want to impose structure on the typically very complex task of designing a user-computer interface, so the design can be divided into manageable pieces, each of which can be dealt with in a systematic, rigorous and at least partially quantitative way. We believe this will help make User Interface Design more of a science and less of an art, and lead to improved design.

The actual process of designing a user interface can be accomplished as four major steps, which we call the conceptual, semantic, syntactic, and lexical design steps. Each step can be dealt with in sequence, one after the other, with an occasional reexamination of a previous step. We call these four steps a design framework.

The Design Framework

The conceptual design is the definition of the key application concepts which the user of the interface must understand in order to use the system. For a simple text editor, the key concepts are files, lines of a file, and operations (add, delete, move) on lines. The conceptual model, as in this case, typically defines objects, relations between objects (a line is in a file), and operations on the

objects, and sets the stage for the semantic design of the user-computer interface.

The semantic design deals with the functionality of the system to be accessed via the intermediary of the user interface. The user performs certain actions, calculations/processing ensues, and information is presented to the user. At the semantic design level we are concerned only with the meanings of the inputs, the processing, and the outputs: we are not concerned with the form or the sequence of the inputs and outputs.

The syntactic design deals with the sequence of the inputs and outputs. For the input, sequence is akin to grammar--the rules by which sequences of words in a language are formed into legitimate sentences. The types of words in an input sentence are typically commands, quantities, names, coordinates, or arbitrary text. As in English, the words are the units of meaning in the input and cannot be further decomposed without losing their meaning. To include the spatial domain as well. Therefore the output syntax includes the 2D or 3D organization of a display as well as any temporal variation in the form. The "words" in the output sequence, by analogy to the input sequence, represent the units of meaning being conveyed from the computer to the user. The units of meaning are often conveyed graphically as symbols and drawings made up of lines, curves, and points rather than as words made up of letters.

The lexical design determines how words in the input and output are actually formed from the available hardware capabilities. For input, this involves designing the interaction techniques for the application. An interaction technique is a way of using a physical input device (tablet, keyboard, mouse, etc.) to input a certain type of word (command, value, coordinates, etc.). For example, some of the interaction techniques for command specification are selection from a menu with a light pen or with a cursor controlled by a mouse, typing of the command name on a keyboard, and speaking the name of the command into a speech recognizer.

For output, lexical design means forming the symbols and shapes which are to be presented to the user, using the available hardware lexemes. For text output, this reduces to selecting text attributes such as font, size, color, background color: the spelling (i.e., combination of hardware lexemes, the character set) of words is already defined in the dictionary. In other cases, such as situation displays, the symbols used must be designed and composed from lexemes such as lines and other graphics primitives, and the symbols must be assigned attributes such as color, intensity, linestyle and size.

The nub of this four-level framework for design are found in formal language theory; the framework has been successively refined and reported in a series of papers

[FOLE74, FOLE78, FOLE80, FOLE81b]. We have worked/are working with this framework in several ways: the organization of design principles, the evaluation of existing user-computer interfaces, the evaluation of interaction techniques (which are the lexical-level design of the input), the formal specification of the syntactic and lexical design of input and output, the calculation of metrics of "goodness" based on the formal specification, and the design of an "abstract interaction handler" to remove much of the syntactic and lexical design from the application program.

Organizing Design Principles

The past ten years have seen several user interface designers setting forth their design principles [BENN76, BRITT77, ENGE75, HANS71, WALL76] in the form of general objectives and specific do's and don't's. These papers plus personal experience form the knowledge base available to most designers. Often the criteria are soundly-based: a useful start in developing tools for designers is to organize the principles, showing how they apply at the conceptual, semantic, syntactic, and lexical design levels. This process has been partially completed, as reported in FOLE81b, for principles dealing with feedback, error correction, response time, consistency, and display structure.

Evaluating User-Computer Interfaces

Given an organized set of design criteria, it is possible to perform a systematic evaluation of existing user-computer interfaces by a combination of watching others use the interface and learning to use the interface oneself. In this process it is critical to note idiosyncratic features of an interface when they are first encountered, lest one adjust to the features. Two such evaluations have thus far been conducted: the first [HERB80] of DIDS, the Decision Information Display System used by the federal government for policy studies; the second [BLE81] of SEEDIS, the Socio-Economic Environmental Demographic Information System developed at Lawrence Berkeley Labs. A third evaluation will be of a new user-interface design, prior to its implementation, for Battelle Northwest Labs' ALDS (Analysis of Large Data Sets) system.

Evaluation of Interaction Techniques

Recall that an interaction technique is a way of using a physical input device to input a word, and hence is the lexical level input design. In FOLEB1a we have described and organized the interaction techniques by their purpose, which can be to make a selection, designate a position, orientation, or sequence of positions and orientations, input a value, or input a character string. A number of germane human factors design issues have been identified for the techniques by drawing on the literature and the

guidelines mentioned above. Nine experiments dealing with interaction techniques are also critically reviewed. A method of interaction technique diagrams is created, to aid in understanding, analyzing, and documenting the techniques and experiments. A diagram shows the cognitive, motor, and perceptual steps which the user of a technique performs. The report is meant as a guide to aid designers in selecting appropriate interaction techniques and devices.

Formal Specification and Metrics

The syntactic and lexical designs of a user interface should be describable by formal language tools, in the spirit (but not necessarily in the image) of BNF, regular expressions, and flow expressions. We are developing formal tools for describing both the input and output of a user interface, as well as the relationship between input and output. The input definition deals with concepts such as token types (which are the purposes of interaction techniques, as described above), sequences of tokens, and the binding of tokens to sequences of actions with physical devices. The output definition deals with concepts such as screen areas and their contents, and attributes (such as color, font, and linestyle) of tokens within various areas. Metrics treat issues such as complexity and consistency of syntactic rules, consistency in the use of codings, continuity of visual attention on the display, continuity of tactile motion with the interaction devices, and time required to input commands. The metrics draw upon the guidelines mentioned above.

The designer of a user interface will use the tools to describe the interface. This in itself helps create a more disciplined design environment. In addition, the formal definition will be processed, metrics evaluated, and potential design problems flagged for further attention by the designer. In the long run, the user interface definition

will be input to an interaction handler which will actually implement the user interface.

Abstract Interaction Handler

Writing an interactive application program involves coding the semantic, syntactic, and lexical designs, typically using FORTRAN, PASCAL, or a similar language. There are two problems with this. First, the procedural languages are not well-suited to programming the syntactic and lexical designs. Secondly, it is easy to intertwine the code which implements each of the three levels, making later changes to any of the levels difficult. The abstract interaction handler is being designed to implement the syntactic and lexical aspects of input, and those parts of the syntactic and lexical output design having to do with interaction, such as menus, prompts, and error messages.

This approach allows much of the user interface to be changed by modifying the interface definition made available to the interaction handler rather than by reprogramming. It will be possible to use two completely different user interfaces, such as menu driven and command-language driven, with the same application program, and to "fine-tune" the details of a given user interface. Within the interaction handler, syntactic and lexical level designs will be separated, so that one can be easily changed without affecting the other. A preliminary design of an interaction handler can be found in FELDS1.

References

- BENN76 Bennett, J., "User-oriented Graphics Systems for Decision Support in Unstructured Tasks," Proceedings of ACM/SIGGRAPH Workshop on User-Oriented Design of Interactive Graphics Systems, Pittsburgh, PA., October 1976, pp. 3-11.
- BLES81 Blesser, T., P. Chan Mei Chu, "A Critique of the SEEDIS User Interface," The George Washington University, Institute for Information Science and Technology Tech. Report GWU-IIST-81-04, March 1981.
- BRIT77 Britton, E., "A Methodology for the Ergonomic Design of Interactive Computer Graphics Systems, and Its Application to Crystallography," University of North Carolina at Chapel Hill, UNC Report No. TR-77-011, November 1977.
- ENGE75 Engel, S., and R. Granda, Guidelines for Man/Display Interfaces, IBM Poughkeepsie Laboratory, TR 00.2720, December 1975.
- FELD81 Feldman, M., "Preliminary Design of an Abstract Interaction Handler," The George Washington University, Institute for Information Science and Technology Tech. Report GWU-IIST-81-06, Washington, D.C., 1981.
- FOLE74 Foley, J. and V. Wallace, "The Art of Natural Graphic Man-Machine Conversation," Proceedings IEEE 62(4), April 1974, pp. 462-470.
- FOLE78 Foley, J., "The Human Factors-Computer Graphics Interface," Proceedings of Symposium on Human Factors and Computer Sciences, Computer Systems Technical Interest Group, Human Factors Society, June 1978, pp. 103-114.
- FOLE80 Foley, J., "The Structure of Command Languages," in R.A. Guedj, et al., eds., Methodology of Interaction, North-Holland, Amsterdam, 1980, pp. 227-234.
- FOLE81a Foley, J., V. Wallace, and P. Chan, "The Human Factors of Interaction Techniques," The George Washington University, Institute for Information Science and Technology Technical Report GWU-IIST-81-03, Washington, D.C., March 1981.
- FOLE81b Foley, J., "A Methodology for the Design and Evaluation of User Computer Interfaces," The George Washington University, Institute for Information Science and Technology Technical Report GWU-IIST-81-05, Washington, D.C., March 1981.

- HANS71 Hansen, W., "User Engineering Principles for Interactive Systems," Proceedings 1971 Fall Joint Computer Conference, pp. 523-532.
- HERB80 Herbert, T., "Evaluation of the User-Computer Interface Design of the Domestic Information Display System," The George Washington University, Department of Electrical Engineering and Computer Science Technical Report GWU-EECS-80-07, Washington, D.C., 1980.
- WALL76 Wallace, V., Summary of "Conversational Ergonomics" Session, ACM/SIGGRAPH Workshop on User-Oriented Design of Interactive Graphics Systems, Pittsburgh, PA., October 1976, pp. 121-122.

Psychological structure in information organization and retrieval:
Arguments for more considered approaches,
and work in progress.

George W. Furnas
Computer-user Psychology Research Group
Bell Laboratories, Murray Hill, NJ

Any given artificial storage and retrieval system forces structure on the information stored within it. Psychologically, however many kinds of structures exist for the representation of information, and each has domains where it is well suited and domains where it is at best misfit. The motivating assumption here is that, if one wishes to make information systems humanly accessible, more serious consideration is needed of the variety of representations characterizing human knowledge, coupled with the necessary invention of new compatible retrieval interfaces.

A textile dyer would no doubt be exasperated by a menu-driven, or even key word, specification of colors. Our knowledge of color space argues that adjusting three knobs, or perhaps moving a light pen on a graphics screen would probably be much better. In contrast, asking zoo visitors to access information about individual animals by this same three-knob technology would be ridiculous. Menus or keywords would be very appropriate. The domain of animals has a very different structure than does that of color, and to use the same retrieval system for the two is a mistake.

Not much experimental evidence exists regarding implications for computer access, but from the standpoint of reflecting psychological similarity, recent work by Pruzansky, Tversky and Carroll (1980) emphasizes the diversity of appropriate representations. Using currently available scaling procedures in a large survey of categories, they typically found the domains to differ strongly in the relative suitability of tree and multidimensional structures for capturing people's similarity judgements.

There are of course even more representational structures than the two investigated by Pruzansky, et al. From the context of similarity scaling alone, one might mention, in addition to multidimensional spaces and hierarchical clusterings, additive trees, more general graphs, factor-analytic structures, additive clusterings, etc. These structures differ in many ways, including continuity, contingency constraints on structural components, complexity, and symmetry. All of these properties presumably affect representational adequacy.

Scaling techniques, among others, can help to identify psychological adequacy of representations. but in constructing retrieval systems, a further issue arises: How can any of the variety of possibly appropriate representational structures be accessed? Hierarchical tree structures lend themselves to classical menu-tree schemes, and multidimensional configurations with suitable properties (e.g. low number of dimensions, separability?) may perhaps be accessed by various analog input devices. But what of other types of structures, especially as we seek richer structural representations?

Thus cognitive considerations motivate the search for nonstandard database interface solutions... new structures, and new access processes. The work presented here represents a simple ongoing effort in that direction. It basically involves a generalization of tree structures, and of the corresponding familiar menu access mechanisms.

Standard menu systems present a screenful of choices subdividing the domain of a database. The user makes a selection from these, resulting in a new set of more detailed selections, further subdividing the selected set. A sequence of choices from a succession of menus eventually brings the user to some final target item. Typically, the menus are organized into trees. That is, there is usually only one sequence of choices that will arrive at any given target. While some systems have exceptions to the unique path rule, these tend to be infrequent, and certainly not essential to the character of the system.

Note that in menu trees, there are many choices, a whole menu full, presented at each step when moving down through the structure. There are occasions, however, when one must move back upward in generality, as in recovering from a mistake or changing targets in mid-search. Then, unlike when moving downward, there is no choice given: Trees have many "down" choices at any point, but only one "up". The concept being explored here revolves around allowing menus for upward choices, as well as the usual downward ones.

The psychological motivation goes as follows: Consider a given node, or point of menu presentation in the structure, to represent a conceptually defined class of possible targets. A given conceptual class can certainly contain many different subordinate classes, enumerated in the downward menu, but often in rich domains the class can also be contained in many superordinate classes. A traditional tree representation is forced to organize on the basis of only one superordinate at each level. In so far as these different superordinates may each be useful in different circumstances, this psychological organization should be reflected in the access structure, by giving users choice when moving to superordinate levels.

Imagine. for example, one had a computerized system for retrieving cooking recipes that was being used to plan a meal. Imagine further that the user had proceeded down to a screenful of choices about types of salad (CAESAR, SPINACH & MUSHROOM, etc.). but had just decided after all, against any salad for the meal, and was ready to retreat back up the structure to other categories of choices. Conceivably, the user would have been interested in an alternative in the form of some other cold food, say cold cuts instead of salad, so that a superordinate of COLD FOOD would be appropriate in the structure. Alternatively, it might have been that the user wanted some other vegetable dish, so that a VEGETABLE node would have been the most useful superordinate. Or perhaps the user wanted a different early course for the meal, say soup instead of salad. Thus, any of several superordinates (COLD FOODS, VEGETABLE DISHES, EARLY COURSE DISHES) might have been what the user wanted. Why not give the user exactly such a choice, in an Up menu from the salad node, in addition to the typical Down menu? If the user's head prominently figures a certain form of representation, externalize it in the organization of the data, and take advantage of it in the access mechanism.

We are in the midst of exploring the concept of up/down menu (MUD) systems on a small artificial data base of a few hundred target items. There are a number of implementation choices that require research, most notably regarding how to construct the MUD structures: In using normative categorization data, various verification and "garbage collection" ideas must be invoked to ensure that links exist everywhere they are appropriate, and nowhere else. We currently ask subjects to construct "isa" networks by repeatedly nominating successive superordinates from each node, and then use frequency thresholds on nodes and links produced across subjects.

When other subjects are then allowed to use the MUDs, several more profound issues arise. A necessary result of having multiple Up choices is that Down choices are not always partitions of the conceptual class encompassed by a node. The consequence that that some choices overlap is of mixed advantage. Under some circumstances it allows subjects the benefit of approaching a target with different interests in mind or with a different psychological "set," but it can also mean that subjects must not only decide whether a given choice will lead to their target, but weigh the relative merits when several reasonable choices exist. Another issue is that MUD structures lack the systematic traversal algorithms that trees have. Thus it is more difficult to be exhaustive, i.e. to make sure all nodes have been seen at least once, and efficient, i.e. to avoid unnecessary repetitive viewing of nodes. Circumstances exist where these considerations might be important. A third issue is that the class of targets actually subsumed by any downward choice is constant, while the users interpretation of the choice can be effected by the history of superordinates just passed through. In a tree, there is only one possible ancestral history, so no ambiguity arises, but not so in a MUD structure, so users can interpret a choice variably, due to the different emphases of different superordinates.

Some issues also arise in working with MUDs that are perhaps even more relevant to tree structures. Transitivity of class inclusion is critical to any system based on conceptual hierarchy. High level choices require inferring the targets subsumed under intermediate level nodes. Intransitivity can foil this: Suppose one is looking in a lay person's botanical guide for Scrub Oaks which are classified under OAKS, and that OAKS are in turn classified as TREES. The problem is that Scrub Oaks are not popularly considered trees (rather, say shrubs). This lack of transitivity, due perhaps to fuzzy classification systems, would lead one away from a correct choice of TREES in the pursuit of Scrub Oaks. MUD structures have an advantage over menu trees since they can allow other routes to Scrub Oaks that are perhaps free from intransitivities.

While this work represents only one modest example of exploration of more diverse psychologically motivated structures, we believe that efforts like it can lead to systems of greater help to human users.

The Nature of User-Generated
Commands for Interacting with a Computer

Mark D. Jackson

Judith E. Tschirgi

We describe the results of an experiment investigating user conceptions of a natural language for interacting with a computer information system. Novice and experienced computer users performed text editing and information retrieval tasks using a simulated interactive system. For each task, a script or sequence of actions was presented to the user. At each step, users read a description of an action, such as correcting errors in text or selecting a page of information to view, and typed a command that they thought was a natural request for the action. If their command was inappropriate, they were asked to reword their attempt; otherwise there were no constraints on their input.

A diverse set of command terms was generated by both groups of users, with few actions eliciting common terms from within or across groups. Novices used more English-like command formats whereas the experts followed computer dialogue conventions learned from experience. When correcting commands, novices were more likely to use strategies applicable to normal conversation. In general, the responses generated

by our users under instructions to produce "natural" commands depended critically on the user's experience.

We have shown that computer experience affects users' spontaneous approaches to interacting with a computer. Our results suggest that no single command syntax or set of lexical items will be consistent with the expectations of all users without additional training. Thus, future research must determine the characteristics of easily learned person-computer dialogues that are "natural" across all levels of experience.

A Conceptual Approach to Natural Language Fact Retrieval

Janet L. Kolodner
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

1. The problems

person: What's been going on in the world recently?
computer: The last hostage was finally released from Iran.
person: I thought the hostages were released all at once.
computer: She wasn't really one of the hostages, but was
arrested later when she traveled to Iran as a
journalist to cover the Iranian revolution.
She's been referred to as the 53rd hostage.
person: Why was she arrested?
computer: Trumped-up espionage charges.

Suppose we wanted to build an intelligent fact retrieval system such as the one above. What would that require? It would have to be able to deal intelligently with a human user, giving answers containing not only the appropriate information, but also the right amount of information. It would have to be able to analyze the intent of a human question or response, figuring out what the questioner really wanted to know. The system would also have to be able to search its memory in a smart way, so that as the memory grew, it would still respond in a reasonable amount of time.

There are three major problem areas to be addressed in designing such a system:

1. Interfacing with the user: analyzing his natural language questions, and deriving search keys from them
2. Memory search
3. Memory organization and maintenance

These problems cannot be solved independently of each other. The organization of memory constrains the types of retrieval and updating processes the memory can have. On the other hand, memory organization, and therefore procedures for adding information to memory, must be designed based on retrieval requirements. Similarly, memory's organization and content, and the relationship between items and categories in memory should be taken into account in interpreting the intent of user questions.

The CYRUS system has dealt with aspects of all three of these problems. CYRUS has a long term memory which was designed to store information about important political dignitaries. It has been used to store and retrieve information about former Secretaries of State Cyrus Vance and Edmund Muskie. CYRUS automatically adds new information to its memory, maintaining good memory organization in the process. It can be queried in English, and uses retrieval strategies and knowledge about the organization of its memory to search for answers. A successor to CYRUS, TED, will keep track of events in the life of Ted Turner, a celebrity, sports figure, businessman, and broadcasting figure.

The remainder of this paper will outline some of the problems involved in designing a fact retrieval system which will communicate effectively with people. Interactions between the interface, memory search, and memory organization will be described. It will also outline the solutions to these problems, as implemented in CYRUS and described in Kolodner (1980).

In considering these problems, we will assume a memory organized by conceptual categories, with events indexed and sub-indexed in those categories by their salient features. Thus, memory processes will

manipulate conceptual information, or the meaning of the data in the memory, and will not be concerned with the words used to express those concepts.

2. Retrieval requirements

2.1 Choosing a category for search

Searching a memory organized in categories requires specification of a category or categories to be searched. Consider, for example, the following question:

(Q1): Mr. Vance, when was the last time you saw an oil field in the Middle East?

If "seeing oil fields" were one of memory's categories, then this question would be fairly easy to answer. "Seeing oil fields" would be selected for search. If it indexed an episode in the Middle East, that episode could be retrieved from it. Similarly, if "seeing objects" were a memory category, it could be selected for retrieval and events in the Middle East and events at oil fields could be retrieved.

If neither of these categories existed, however, a category for search would have to be chosen. We can imagine the following reasoning process being used to do that:

A1: An oil field is a large sight, perhaps I saw an oil field during a sightseeing episode in the Middle East.

Using information about episodic contexts associated with "large sights", a "sightseeing" category can be chosen for retrieval. Its contents can be searched for an episode at oil fields in the Middle

East. If the sightseeing category organized its episodes according to the type of sight and its part of the world, and if there had been an episode in the Middle East at an oil field, then "a sightseeing episode at an oil field in the Middle East" could be retrieved.

The problem of choosing a category for search is both an interface problem and a search problem. Search requires specification of a category to be searched. For a very complex data base, however, we cannot expect a user to know all of memory's categories. Nor can we expect that every natural language question asked of a data base will specify a category for search.

In CYRUS, this problem is solved by associating with each concept in memory the categories it is related to. Thus, the concept "large sights" has "sightseeing" associated with it, while "international contract" has the category "political meetings" associated with it. In the first step of the retrieval process, the conceptual representation of the question (produced by a conceptual analyzer) is checked to see if it already specifies a category for search. If not, contexts are chosen from among the categories associated with each of the question components.

2.2 Non-enumeration

One of the most important problems to address in designing an interactive retrieval system is the following:

Retrieval should not have to slow down as memory grows.

This requirement constrains both the retrieval processes and the memory organization. In terms of the retrieval processes, it requires the following:

Retrieval from a category must be able to happen without enumeration of the category.

In fact, this interface problem depends on both memory organization and retrieval processes for a solution. If categories cannot be enumerated, then there must be some other way of searching a category. This can be done by indexing items intelligently in categories, and then by specifying and following appropriate indices during retrieval.

This method of retrieval brings up special problems. Retrieval is easy if a question specifies features which are indexed. This is not always the case, however. Two solutions to this problem have been implemented in CYRUS -- automatic generation of plausible indices, and search for alternate contexts.

2.2.1 Index fitting and generation of plausible features

Just as we cannot expect a user to know all of memory's categories or to specify a category in his question, we cannot expect him to know memory's indexing scheme. Thus, features specified in a question might not correspond to features indexed in memory. In that case, given features must be transformed into indexed features.

Inferring indexed features is a way of directing search within a memory category without enumerating the category. Generated features can be followed to find the target item in the category. In addition, there must be a way of recognizing that two different descriptions refer to the same item. One way to do that is by transforming one description into the second one.

Continuing with the example above, suppose sightseeing episodes were not organized in a category according to the type of sight or by

their place in the world. In that case, the following elaboration of the initial retrieval specification might be appropriate to answer the question:

A2: Which countries in the Middle East have oil fields? Iran and Iraq have oil fields, and Saudi Arabia does. ...

If sightseeing episodes are organized according to the country they took place in, then elaborating on "the Middle East" and specifying particular countries in the Middle East would enable retrieval of episodes that took place in each of those places. Instead of searching for "sightseeing at an oil field in the Middle East", search for each of the more specific episodes "sightseeing at an oil field in Iran", "sightseeing at an oil field in Iraq", etc. could be attempted.

The process of transforming given features into indexed ones is called index fitting. Index fitting is done in CYRUS by component-instantiation rules. These rules use information about components in context to infer additional features of a specified item. The nationality of participants in a political meeting, for example, is known to correspond to the sides of the contract being discussed at the meeting. Given the participants in a meeting, that information can be used to infer aspects of the meeting topic. Component instantiation rules generate plausible features for a targetted item. These features correspond to indices which should be traversed to retrieve that item from memory.

2.2.2 Alternate context search

Elaboration of plausible features is only one way of directing search, and it is not always successful. Suppose, for example, that

there was not enough information to narrow a search key to an easily enumerable (i.e., small) part of the data base. In a memory where records refer to other contextually related records, it might instead be appropriate to search memory for an alternate, more retrievable context. In other words, retrieval can proceed by searching for a related context which (1) might be more retrievable than the target item, and (2) might refer to the item targetted for retrieval.

Since CYRUS' memory is organized in event categories, alternate context search in CYRUS corresponds to search for an episode related to the targetted event. Since sightseeing in the Middle East would have had to happen during a trip to the Middle East, retrieving a trip to the Middle East could aid retrieval of an appropriate sightseeing experience. Thus, the following reasoning would also be appropriate to answer the question above.

A3: In order to go sightseeing in the Middle East, I would have had to have been on a trip there. On a vacation trip, I wouldn't go to see oil fields, so I must have been taken to oil fields during a diplomatic trip to the Middle East. Which countries might have taken me to see their oil fields? Saudi Arabia has the largest fields, perhaps they took me to see them. Yes, they did when I was there last year.

Why does it seem reasonable to search for "trips" when a "sightseeing" episode should be retrieved? How can search for alternate events be constrained? Only alternate contexts that might be related to an event targeted for retrieval should be searched for.

In general, for search to be constrained to relevant contexts, memory categories must hold generalized information concerning the relationships of their items to items in other memory categories. In CYRUS, alternate context search is facilitated by three things:

1. knowledge of the usual relationships between event categories
2. a set of context construction rules for constructing a new context based on that knowledge
3. a set of search strategies for directing search for the target event within the context of the alternate event

Thus, CYRUS knows about the usual relationship between sightseeing and trips, how to construct a trip context based on a sightseeing context, and how to search the sequence of events of the trip to find a sightseeing experience once an appropriate trip is found.

2.3 Maintaining a conversational context

Maintenance of a conversational context is necessary for resolution of ambiguous references, anaphora, and pronominal reference. Suppose, the question above were followed in conversation by the following one:

(Q2): Did you talk to the workers there?

In order to understand what "there" means, the answer to the previous question must be consulted. In order to understand which workers are being talked about, the context of "visiting oilfields", plus knowledge about oilfields themselves must be used.

Maintenance of a conversational context can also constrain memory search. Often, it is necessary to search only the context of the answer to the previous question to find an answer to the current one. In the example above, for example, only the events involved in Vance's visit to the oilfield in Saudi Arabia need be searched for an answer. If the previous context is maintained, it can constrain search to that episode only, so that all of memory does not have to be searched.

2.4 Summary of retrieval

The retrieval process described can be seen as a process of reconstructing what might be true, and checking memory to make sure it indeed was. To retrieve an episode of "seeing oilfields", a hypothesis was made about the type of event it might have been (sightseeing), where it might have happened (Iran, Iraq, Saudi Arabia, etc.), and what else might have been going on at the time (a trip).

Judging from this example, the process of retrieval requires at least the following processes:

1. selection of a category for search
2. search within the category for the targeted event
3. elaboration on the specification of the event to be retrieved
4. search for episodes related to the target event

3. Requirements on the memory organization

The ability of memory to support retrieval without enumeration is also dependent on the memory organization. The traditional solution within computer science to the non-enumeration problem is to index items within categories. An event should be indexed in a category by those of its features that are salient to the category. In that way, specification of an indexed feature will enable retrieval of items with that feature without enumerating the whole category.

If memory categories are heavily indexed by salient features, retrieval processes will have a large selection of features to specify, any of which might specify a target event. The retrieval process will

be made easier since the easiest elaborations can be attempted first.

The richer the indexing, however, the more space is needed for storage. Indexing must be controlled so that memory does not grow exponentially. In CYRUS, similarities between events are used to control indexing. Memory keeps track of the similarities between events within a category, and limits indexing to the differences between events. Thus, if almost all the events in a "diplomatic meetings" category are with foreign diplomats, indexing them according to the occupations of their participants would be redundant and therefore unnecessary. It would not divide the category into significantly smaller parts. If, however, one of those meetings were with someone other than a foreign diplomat, indexing the meeting by that feature would differentiate it from other events in the category. In fact, the similarities which constrain indexing correspond to the generalized information necessary for retrieval.

Finally, a memory for events should maintain itself. This means that the process of selecting indices should be automated. It also means that events must be sub-indexed within the sub-categories that are formed when multiple events are indexed in the same way. Otherwise, the sub-categories would have to be enumerated. This places another requirement on the updating processes. In order to constrain later indexing, and in order to guide the retrieval strategies, the automatic updating process must also keep track of the similarities between events in each newly-created sub-category. If we don't want retrieval to slow down as new events are added to memory, then memory must be able to maintain its organization, creating new conceptual categories when necessary and building up required generalized information. CYRUS does

this through a series of organizational strategies.

Another aspect of maintaining memory's organization involves monitoring memory search. More frequently requested information should be more accessible than less frequently requested information, and more recently accessed information should be more accessible than less recently accessed information. This involves both reorganization of memory taking frequency of access into account and restructuring the organizational strategies themselves, so that more frequently asked for types of information will automatically be organized for accessibility as they are added to the data base. This, and other memory maintenance problems which have not been described here, are being addressed in current and future research.

Psychological Investigations of
Natural Command and Query Terminology

Thomas K. Landauer
Susan T. Dumais
Computer-user Psychology Research Group
Bell Laboratories, Murray Hill, NJ

It is frequently asserted that unsophisticated users would find computer systems more congenial if communications with them were to employ more "natural" words. In a series of empirical studies, we have (1) developed a method for identifying natural command words for a particular task, (2) tested the value of the resulting natural command lexicon in the initial stages of transfer from manual to automated task performance, and (3) induced people to form "natural" data queries and analyzed the language they used.

Identification of "natural" command terms. Twenty-two students in secretarial schools and twenty-six high school students with typing skills were given manuscripts with author's marks. The author's marks indicated a variety of desired corrections corresponding systematically to the kinds of changes that are accomplished in manual or computer text-editing operations. The students were asked to write instructions to another typist, who did not have the author's marks, specifying what was to be done to the manuscript. This method produced verbal descriptions of actual editing operations (e.g. "take out the word the") as contrasted to description of the author's marks (e.g. "crossout") or goal (e.g. "fix the spelling"). Among noteworthy resulting observations were the following:

- (1) There was little agreement on word use; e.g. the three most frequent operational verbs used accounted for no more than 33% of descriptions of any one correction,
- (2) The words used were not like those commonly employed by computerized editing systems, e.g. the verb "delete" was never used, and
- (3) Unlike many computerized text-editing systems, students and secretaries tended to use different words to describe operations on characters and blanks, but the same words to describe similar operations on whole lines and line-internal strings (e.g. "change 'string a or line a' to 'string b or line b'").

Testing the value of natural command terms for initial learning. We devised a set of miniature text-editing systems, each consisting of only append, delete, and substitute operations plus start and stop commands. For one version, the verbs used in

the operation commands were "append", "delete" and "substitute", terms often used in computer text-editors. For another, they were the verbs most frequently used by secretaries and typists to describe the required action, "add", "omit", and "change", respectively. A third variant used randomly chosen English verbs, "cipher", "allege", and "deliberate" as a baseline control for lexical naturalness. In addition, the text-editors varied (a) with respect to whether the command verb was to be spelled out or abbreviated to its first letter, and (b) with respect to whether the same command word applied to both line-internal strings and whole lines (e.g. "omit /a/" for within - and "omit" for whole-line) or used different command words (e.g. "change /a/" for within-line and "omit" for whole-line). Forty-eight secretarial and typing students each spent about two hours studying an introductory self-instructing manual and simultaneously doing a series of on-line learning and test exercises. The manuals varied only in necessary ways (essentially only in command names) and as little extra help as possible was provided.

The main results of interest were as follows: (1) The time to perform test exercises was not significantly influenced by command name variations; subjects performed as well when they were learning to "allege", "cipher", and "deliberate" as when they were learning to "add", "omit" and "change". However, a post-session questionnaire revealed some subjective preference for the more familiar terms. It is also important to note that the subjects were learning a very simple system with very few terms, and that they were not required to remember the terms over substantial periods. It is possible that "natural" terms would be advantageous in larger lexicons or when long-range recall was necessary. However, natural words do not appear to provide substantial benefit during the highly critical first few hours of introduction to the new and exotic computer aided text-editing environment, as one might have expected and/or hoped. (2) Abbreviated command names were slightly more time-consuming to use at first, but became significantly less so after some practice. (3) In this case, at least, the use of different command names for whole-line and within-line operations resulted in better performance than using the same name for both. This is contrary to subjects' usage in spontaneous descriptions. We hypothesize that the requirement to use different syntactic constructions in our editors was responsible; that differing command words make it easier to learn and use differing constructions even if the operations are naturally thought of as similar.

Characteristics of natural data specifications. Three hundred and thirty-seven college students tried to specify verbal objects. They were given a list of items like "newsweek", "Empire State Building", etc. and asked to try to specify each so that another student or (in other cases) a computer would

respond with the provided word. There were no restrictions as to the form or content of the descriptions (except, of course, that they could not contain the target item).

Among interesting characteristics of the response were these: (1) Students rarely used boolean expressions more complicated than simple conjunction. (2) Specification by exclusion (e.g. "a popular weekly newsmagazine other than Time") was very infrequent despite the intentional inclusion of items that easily admitted of such specification. (3) The most common specification techniques were simple lists of positive attributes or a single immediate superordinate, followed by a list of attributes (e.g. "a tall building in New York located on 34th Street and 5th Avenue"). (4) Specifications were often very vague and depended heavily on presuppositions about preferred responses of the target person or system (e.g. "a tall building in New York", a specification that apparently assumes that one member of a large class will be known to be most representative or most dominant and will be given in the absence of further specification).

We have no evidence as yet as to whether systems allowing "natural" query specifications would be easier to use. However, it does seem apparent that the use of more precise expressions cannot be expected without special, perhaps difficult, training.

ORGANIZING MEMORY FOR USE IN UNDERSTANDING

by

Michael Lebowitz - Columbia University

1 Introduction

Episodic memory plays an important role in the understanding of natural language. It can be used to provide context for top-down processing, to determine the segments of a text that should be focused upon, situation-dependent defaults, and so forth. While this should come as no great surprise, it is the case that most of the work relating memory (in the form of databases) and language understanding has emphasized the utility of natural language front-ends for database query ([Harris 78, Kaplan 77, Woods and Kaplan 72], for example), rather than the ways that memory can be used in language processing. Furthermore, what work there has been on using memory for language processing has been in the form of question answering, ignoring entirely the crucial issue of using existing knowledge in memory to help acquire more information. The use of memory in the process of reading text for the purpose of updating memory - and the effect this has on memory organization - is extremely important, and is the issue I will address here.

In the course of this brief presentation I will be using examples from a computer model that is concerned with the relation between language and memory. IPP (the Integrated Partial Parser), written at Yale, is able to read news stories about terrorism and record them in a coherent memory. It makes generalizations that help organize the memories of the events described and are used to assist in later processing. IPP is fully described in [Lebowitz 80]. A second program, RESEARCHER, is in the early stages of development. It

will be based upon IPP, but will include a memory of a scientific domain, built up by reading technical abstracts. Due to the complexity of the material that RESEARCHER will be reading, the use of memory in the understanding process will be extremely important.

The point that I want to stress here is that the need for applying information from memory during understanding (knowledge acquisition) must be considered while attempting to determine an appropriate memory organization. In the space available here I will give several examples illustrating the need for the application of episodic memory to understanding, and then outline an appropriate memory organization that keeps this use in mind.

2 Why we need to use memory in understanding

The following story is rather typical of those read by IPP.

Figure 1: Attack on kibbutz

S1 - UPI, 7 April 80, Israel

Israeli troops today stormed a children's dormitory in a kibbutz on the Lebanese border to free hostages seized nine hours earlier by gun-blazing Palestinian guerrillas and killed all five raiders.

There are two problems in understanding story S1 that memory can help overcome. The first involves the meaning of the word "stormed", which in this domain can refer to either terrorists attacking a building or government officials counterattacking a group of terrorists. A similar problem arises with "seized", which could plausibly refer to either a kidnapping or a building takeover. The later ambiguity is in fact never resolved in this text. Each of these problems is easily overcome by accessing the proper information from memory, generalizations such as those in the next figure, made after reading earlier stories.

Figure 2: Generalizations about extortion in Israel

Israeli troops carry out counterattacks against terrorists.

Palestinians in Israel engage in extortion by taking places over.

Both ambiguous words in S1 can be resolved by assuming that when relevant generalizations exist, words should be disambiguated so that the new story fits the existing generalizations. The first generalization allows the disambiguation of "stormed" as it is read, using this rule. Similarly, we assume "seized" indicates a takeover, since that corresponds to the second generalization. Had the generalization stated that extortions in Israel were usually kidnappings, then "seized" would have been assumed to refer to such an event.

Notice that we cannot expect a person (or computer program) to be pre-supplied with all the generalizations necessary to resolve problems of this sort. Instead, these observations must be developed by reading (or otherwise learning about) specific events and generalizing from them.

The following story also requires information from memory.

Figure 3: Basques implicit in attack

S2 - New York Times, 24 August 79, Spain

Bombs exploded in a French bank and a French immigration office in northern Spain early today, causing damage but no injuries, according to police.

This story does not specify the identity of the terrorists who set off the explosion described. However, most people with some knowledge of Spain are aware that this was probably a Basque attack. Such a conclusion comes from a previously made generalization about terrorists in Spain.

The next figure shows how IPP handles story S2 when it has existing in memory a generalization that Basques are the attackers in bombings in Spain.

Figure 4: IPP inferring default role filler features

Generalization (BASQUE-GEN) already in memory:

S-DESTRUCTIVE-ATTACK with:

ACTOR	(1)	DEMAND-TYPE	SEPARATISM	<<<-----
		NATIONALITY	BASQUE	<<<-----
METHODS	(1)	AU	\$EXPLODE-BOMB	
LOCATION	(1)	AREA	WESTERN-EUROPE	
		NATION	SPAIN	
RESULTS	(1)	AU	CAUSE-DAMAGE	

*(PARSE S2)

Story: S2 (8 24 79) SPAIN

(BOMBS EXPLODED IN A FRENCH BANK AND A FRENCH
IMMIGRATION OFFICE IN NORTHERN SPAIN EARLY TODAY
CAUSING DAMAGE BUT NO INJURIES ACCORDING TO POLICE)

>>> Beginning final memory incorporation ...

Feature analysis: EV16 (S-DESTRUCTIVE-ATTACK)

RESULTS	AU	CAUSE-DAMAGE
METHODS	AU	\$EXPLODE-BOMB
LOCATION	AREA	WESTERN-EUROPE
	NATION	SPAIN

Indexing EV16 as variant of BASQUE-GEN

Inferring feature ACTOR DEMAND-TYPE SEPARATISM <<<-----
of EV16

Inferring feature ACTOR NATION BASQUE <<<-----
of EV16

>>> Memory incorporation complete

In this example, IPP recognizes that S2 is an instance of a generalization that it has made previously (BASQUE-GEN) and uses that generalization to supply default characteristics of the terrorists. In

particular, IPP assumes, corresponding with the generalization, that the terrorists are Basque separatists. The determination of defaults of this sort is a major use of generalizations. IPP also indexes this event as an instance of the most relevant generalization, so that it can retrieve it later to make further generalizations. I will say more about this last point below.

3 Organizing memory for understanding

Examples such as S1 and S2 place several constraints upon the organization for memory. In particular:

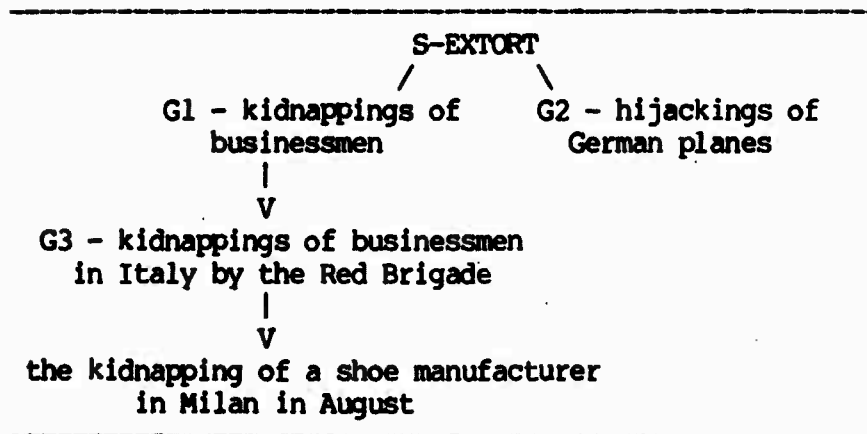
1. It must be possible to access generalizations based on partial information so that relevant information can be applied during understanding, and not just after it has been completed.
2. Many different features of a generalization must provide access to that generalization, so that instances with different relevant features mentioned explicitly can all be identified.
3. Generalizations must lead to memories of actual events so that further generalization can occur.

These constraints suggest a possible memory scheme. This scheme, as implemented in IPP, has several tree-like structures, each consisting of more and more specific versions of generalizations. The generalizations in the tree are used to organize actual memories of events. The trees are associated with high-level knowledge structures that are used to describe events in the domain at an intentional level. (For terrorism these include extortion and attacks on individuals).

A typical tree of generalizations in IPP's memory might look something like the next figure.

A tree of generalizations such as the one in Figure 5 multiple indexing between each generalization and its more specific versions. Normally each

Figure 5: An IPP Generalization Tree



novel feature of a generalization is used as an index for that node in memory. (Some exceptions for common features are mentioned in [Lebowitz 80].) So in Figure 5, generalization G1 could potentially be accessed once a story has been identified as an extortion that is a kidnapping or an extortion with the hostage being a businessman. This kind of identification is exactly what we need to do during the processing of a story so that the remaining information in a relevant generalizations can be used to help processing in the ways indicated above.

The processing scheme that uses such a memory involves identifying the most specific generalizations relevant to a story as it is read, using any features accumulated from the story along with the corresponding generalization index tree. Then the remainder of the story can be interpreted in terms of these generalizations. Further, by having actual events stored under the generalizations, by the time we have finished reading a story we have available similar events that might be suitable for additional generalization.

Similar schemes for organizing memory have also shown to be useful in explaining reminding phenomena [Schank 80] and human memory retrieval [Kolodner 80].

4 Conclusion

Clearly the memory scheme devised for IPP somewhat too simple. For more complex types of data (such as in the scientific domain that will be dealt with by RESEARCHER), memory will clearly have to be more strongly interconnected, resulting in a structure that is more a network than a tree. However, the organization used for IPP indicates how the organization of memory must be appropriate for the process of knowledge acquisition, and not just the retrieval of information.

5 References

- [Harris 78] Harris, L. R.
Natural language processing applied to data base query.
In Proceedings of the 1978 ACM Annual Conference. Association
for Computer Machinery, Washington, D. C., 1978.
- [Kaplan 77] Kaplan, S. J.
Cooperative responses from a natural language data base query
system.
Technical Report, Moore School of Engineering, University of
Pennsylvania, 1977.
- [Kolodner 80] Kolodner, J. L.
Retrieval and organizational strategies in conceptual memory: A
computer model.
Technical Report 187, Yale University Department of Computer
Science, 1980.
- [Lebowitz 80] Lebowitz, M.
Generalization and memory in an integrated understanding
system.
Technical Report 186, Yale University Department of Computer
Science, 1980.
PhD Thesis.
- [Schank 80] Schank, R. C.
Language and Memory.
Cognitive Science 4(3):243 - 284, 1980.
- [Woods and Kaplan 72] Woods, W. A. and Kaplan, R. M.
The lunar sciences natural language information system: Final
report.
Technical Report BBN Report 2265, Bolt Beranek and Newman,
Inc., Cambridge, MA, 1972.

**Artificial Intelligence and Human Factors Engineering:
A Necessary Synergism in the Interface of the Future**

WORKING DRAFT

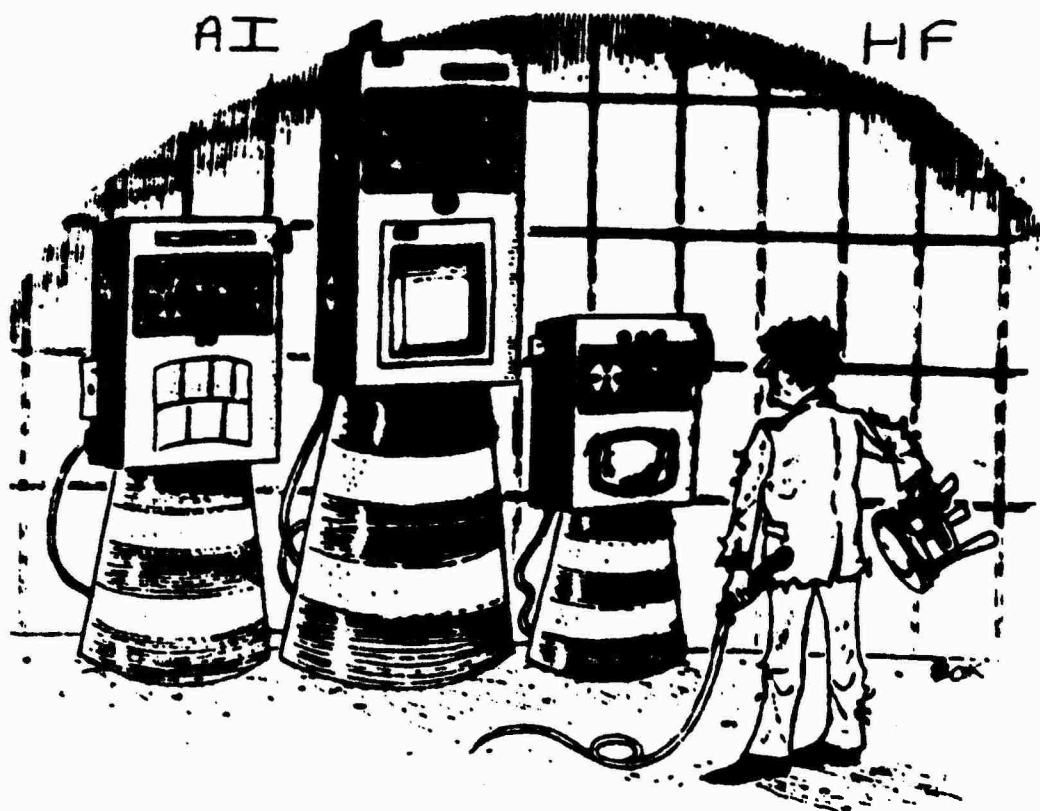
Paul Roller Michaelis and Mark L. Miller

**Computer Science Laboratory
Central Research Laboratories
Texas Instruments Incorporated
M.S. 371, P.O. Box 225621
Dallas, Texas 75265**

ABSTRACT

In the coming decade, a new generation of computer-based systems offers the potential to do for the human mind what the industrial revolution did for human muscle. To realize this potential, we must study sophisticated kinds of software, in which the computer performs tasks previously thought to require human intelligence. We must also study how to organize such hardware/software systems to interact most effectively with their human masters.

TI's Computer Science Laboratory is attempting to construct and evaluate experimental prototypes of such systems. Their design has required unique combinations of talent from diverse disciplines. We are combining expertise from two fields in particular: artificial intelligence and human factors engineering. This talk will illustrate synergistic effects of cooperation between these two fields. Examples will be drawn from current research projects in natural language processing and advanced computer based instruction.



MICHAELIS & MILLER

TABLE OF CONTENTS

1.0 INTRODUCTION

2.0 INTERACTIVE NATURAL LANGUAGE SYSTEMS

2.1 Description of the Problem

2.2 What Human Factors Contributes

2.3 What Artificial Intelligence Contributes

3.0 INTELLIGENT TUTORING SYSTEMS

3.1 Description of the Problem

3.2 What Human Factors Contributes

3.3 What Artificial Intelligence Contributes

4.0 CONCLUSION

5.0 REFERENCES

1.0 INTRODUCTION

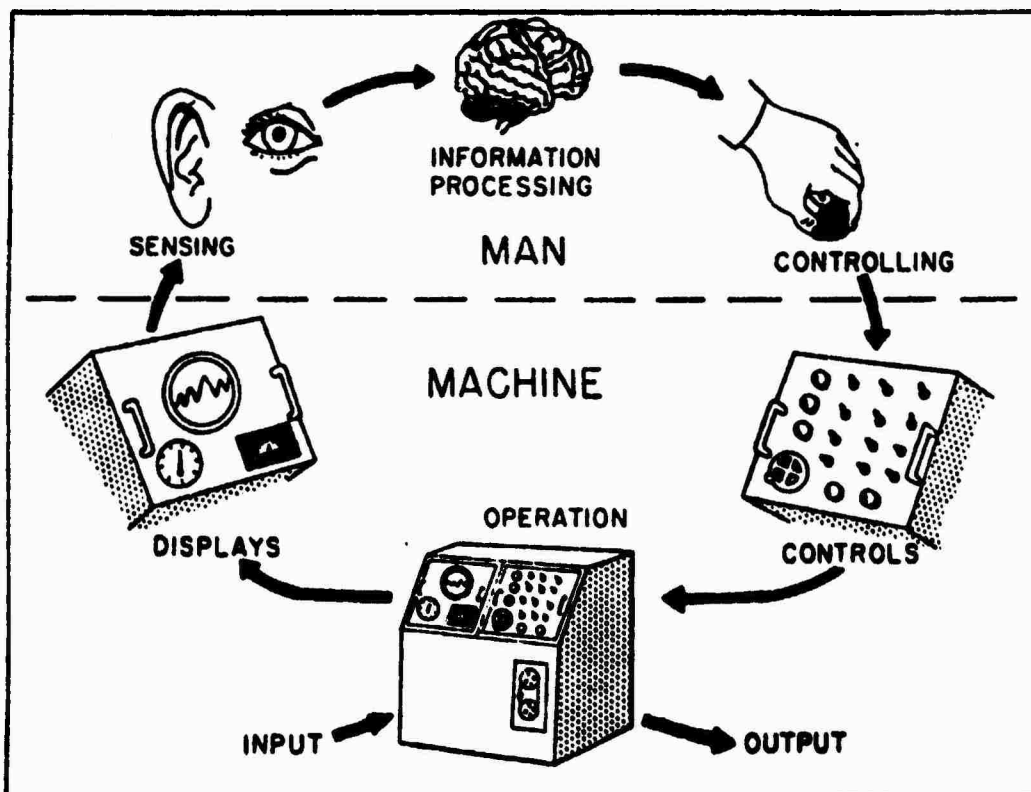
People will have trouble performing a physical task if the demands of the task exceed their physical capacities. To many of us nowadays, that seems like simple common sense. However, it was not until the late 1890's that Frederick W. Taylor made his pioneering studies of how to design jobs and tools so that they more closely match the physical capacities of people. (As an aside, what Taylor studied was shovels and how best to use them.)

The field of human factors engineering had its birth during World War II. The founders of the field recognized that errors can occur in man-machine systems when the man's job in these systems overloads his mental capacities. Before going any further, let's first examine what is meant by "man-machine system." In a man-machine system, one or more of the components is a person, and the person must interact with the machine components. The designs, goals and complexity of these systems vary considerably. Figure 1 shows a schematic of a simple man-machine system.

Show Foil Number -1- Here.
(Man-machine system cartoon from Chapanis, 1965)

During World War II it was found that many errors in human-machine systems, such as airplane accidents due to "pilot error," could in fact be traced to the design of the controls and displays. These are the components of the

THE WORK ENVIRONMENT



system through which the human and machine components exchange information. Researchers such as Alphonse Chapanis and Paul Fitts discovered that certain control and display designs virtually invited even experienced people to misuse or misinterpret them. The solution lay in redesigning the controls and displays so that they operate in manner more compatible with the mental capacities of people.

The TI Computer Science Laboratory develops human-machine systems in which the machine is a digital computer whose software is intended to be (more or less) "intelligent." Efforts to create such artificially intelligent systems have been underway for only a few decades; the founders of the field (e.g., McCarthy [1965], Minsky [1965], and Newell & Simon [1972]) are still active contributors. In even this short time, much has been accomplished. There are systems that can play master-level chess, solve complex integrals, understand and obey commands stated in simple English, speak in a human-like voice, recognize objects in scenes, solve analogy problems, and so on. Central themes, such as the notion of a problem space, means-ends analysis, and heuristic programming have emerged to organize thinking in the field. AI software techniques such as semantic network knowledge representations, augmented transition networks and chart parsers, and production rule deduction systems have gained wide acceptance even as better approaches appear.

The long term goal of this work is to develop "intelligent interactive systems" which do for people's minds what the industrial revolution did for their muscles. Accomplishing this goal requires combining the skills of human factors engineers and AI specialists. The purpose of this talk is to describe the benefits of a synergistic relationship between these two fields. Two research projects currently underway at TI serve to illustrate these benefits.

2.0 INTERACTIVE NATURAL LANGUAGE SYSTEMS

2.1 Description Of The Problem

Chapanis (1975) has demonstrated that interactive natural language dialog is remarkably unruly, with many misspellings and grammatical errors. Although progress has been made in getting computers to process pristine English text, it will be many years before computers will be able to process unlimited interactive natural language dialog.

As our group works toward a system that interacts in true natural language, another project is under way that is oriented toward intermediate results. The goal of this project is to define a human engineered subset of natural language. This subset would retain all of the user-oriented benefits of unrestricted natural language dialog. However, its use would greatly reduce the processing burden that true

natural language interaction places on the computer. This is clearly a goal that can best be accomplished by cooperation between artificial intelligence and human factors specialists.

2.2 What Human Factors Contributes

Ford, Weeks and Chapanis (1980) and Michaelis (1980) reported a series of experiments that were conducted in the human factors laboratory at Johns Hopkins. In these experiments, two-person teams exchanged information over a telecommunications medium in order to solve problems. Half of the teams were rewarded solely for correctly solving their problems. The other half had their correct solution reward diminished for each word token they used. Thus, these latter teams were encouraged to keep their communication as brief and concise as possible. The problem-solving task assigned to the subjects in the Michaelis experiment is typical of the type used in these studies: One team member was given a completely assembled prism-shaped wooden model and was required to assist the other member, who had to build an identical model from the separate parts. In these experiments, the team members were in different rooms. In the Ford et al. study, half the teams communicated by voice and the other half via teletypewriters; in the Michaelis study, all communication was over teletypewriters.

In both studies, there were dramatic and highly significant differences between the two experimental groups. However, it is important to note that problem-solving accuracy was not affected by self-imposed brevity.

Show Foil Number -2- Here.
(Summary of the data presented in the next paragraph.)

Among the significant differences noted in both studies are that the self-limited teams generated, on the average, about one fifth as many word tokens, one third as many word types, and one third as many messages. In a linguistic analysis of the protocols from their study, Ford et al. found that the self-limited subjects used proportionally more nouns (41.9 vs. 26.1%, $p < .001$), fewer pronouns (5.5 vs. 11.9%, $p < .001$), fewer verbs (10.3 vs. 16.9%, $p < .001$), more adjectives (10.3 vs. 10.4%, $p < .001$) and fewer prepositions (8.9 vs. 11.3%, $p < .035$).

Show Foil Number -3- Here.
(Summary of data presented in next paragraph.)

Probably the most interesting finding of these studies is that, on the average, the self-limited teams solved their problems faster than their unlimited counterparts, 14.9 versus 19.3 minutes in the Ford et al. study and 20.5 versus 27.6 minutes in the Michaelis study. This difference was not statistically significant in the Ford et al. study. However, in the Michaelis study, which tested more teams (48

When compared with the unlimited teams, the self-limited teams generated:

- o One fifth as many word tokens.
 - o One third as many word types.
 - o One third as many messages.
-

Mean Percentages of Parts of Speech Used by Teams in the Two Word Usage Conditions. (from Ford, et al., 1980)

Parts of speech	Self-limited	Unlimited	p
Nouns	41.9	26.1	.001
Pronouns	5.5	11.9	.001
Verbs	10.3	16.9	.001
Adjectives	18.3	10.4	.001
Prepositions	8.9	11.3	.035

**Average Number of Minutes for Teams to Solve Their Problems
in Both Experiments and Word Usage Conditions.**

Experiment	Self-limited	Unlimited	p
Ford et al.	14.9	19.3	N.S.
Michaelis	20.5	27.6	< 0.005

vs. 32), the p value was less than 0.005. This is strong evidence that requiring people to be concise does not hurt their ability to communicate; it may even help.

2.3 What Artificial Intelligence Contributes

At this point, natural language specialists in the Texas Instruments AI group became involved. They contrasted the limited and unlimited protocols from the Michaelis study. Their goal was to determine how the dialog limitation might affect the processing burden of natural language computer systems. They were specifically concerned with contrasting the effects on systems that do a syntactic analysis first and then pass the results to a semantic component, versus those which integrate the semantic and syntactic components during analysis.

Pronominal reference and the attachment of prepositional phrases, two stumbling blocks for many present syntactically based systems, occur somewhat less frequently in the limited condition. However, in the limited protocols over one third of the utterances were ungrammatical, while in the unlimited case this was closer to one tenth. They therefore believe that syntax-first approaches will have significantly more problems parsing the limited condition utterances than systems which have less reliance on syntax.

The word types used in the limited condition are virtually a subset of those used by the unlimited users; apparently, many of the words used by the unlimited subjects were not necessary for the solution of the problem. This finding has also been reported in a study of interactive limited-vocabulary dialog (Michaelis, Chapanis, Weeks, & Kelly, 1977), and suggests that the conceptual coverage of the limited protocols is less than that of the unlimited. Therefore, a semantics based system, such as a semantic grammar (c.f. Burton, 1976) or conceptual analyzer (c.f. Schank, 1975), could possibly gain efficiency from the language limitations.

The protocols were also analyzed to examine whether the problem solving strategies used were different between the unlimited and limited conditions. The protocols were classified according to the problem solving strategies used and the ordering of their subgoals. No statistically significant differences were found between the unlimited and limited conditions in the number of teams using the different strategies.

In 38 of the 48 protocols (nineteen in each condition) the subjects used subgoals characteristic of classic means-ends analyses (Newell & Simon, 1972). These teams established two major subgoals of the task, building the triangular sides and building the rectangular base. The order in which these were performed did not significantly

differ between the limited and unlimited conditions.

The ten remaining teams did not have obvious subgoals; six used an approach in which they described the appearance of the model, and the remaining four used a strategy of making small pieces and then connecting these together. Again, no significant differences were found between the two conditions in the number of teams using each strategy.

Show Foil Number -4- Here.
(Conclusions from NLP research)

To summarize the findings thus far in this research effort, human factors specialists found no evidence that the dialog restriction discussed in this paper will hurt the user's efficiency. Indeed, the Michaelis study suggests that the efficiency of the users may actually be improved by well chosen limitations on the interactions. Further, the language restriction could not be shown to significantly change the problem solving strategies used by the subjects. The protocol analyses performed by artificial intelligence specialists suggest that semantically based interactive natural language processing systems might also benefit from this restriction.

Conclusions

From a human factors perspective:

- o No evidence that the dialog restriction hurts people's ability to communicate.
- o No evidence that the dialog restriction changes people's problem solving strategies.

From an AI perspective:

- o Some evidence that a semantically based interactive natural language processing system might benefit from this dialog restriction.

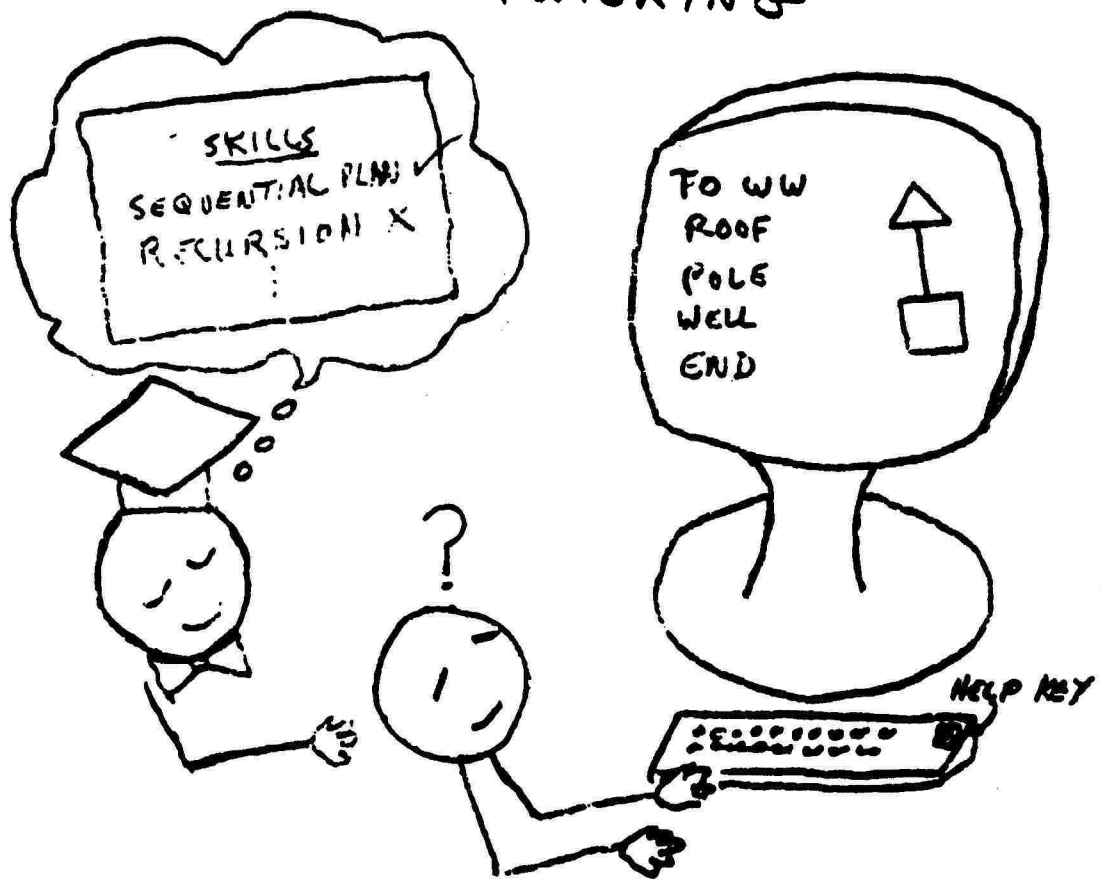
3.0 INTELLIGENT TUTORING SYSTEMS

A second illustration of the AI/HF synergism involves the development of "intelligent tutoring systems" intended to teach elementary computer programming. Such systems represent enhancements over conventional "drill and practice" or "frame-based" multiple-choice branching systems because they incorporate considerable knowledge about the task, the student, and about tutoring per se. The long-term goal is to provide a computer-based educational experience comparable to a one-on-one interaction with an expert human tutor.

3.1 Description Of The Problem

Three systems intended to teach elementary computer programming are examined. The first system, BIP (for "BASIC Instructional Program") [Barr, Beard and Atkinson 1976] serves as a problem solving laboratory wherein students solve programming exercises in the BASIC language. The second system, SPADE-8 (for "Structured Planning and Debugging Environment") [Miller 1979], was developed as an AI exercise. Its purpose was to articulate the underlying programming knowledge involved in writing graphics programs in the LOGO [Papert 1980] language. A third system, currently under development, blends both AI and human factors considerations.

INTELLIGENT TUTORING



BIP is an effective working computer-based instructional system which has been in use by the Navy for some time, both as a research vehicle and as a practical tool for teaching BASIC programming. It serves as a useful representative of previous work on programming tutors. (BIP has been in use in our laboratory for over a year.) BIP is able to individualize the sequence of tasks encountered by a given student, on the basis of a skills model inferred from performance on prior tasks. Hence, not every student encounters every task; and some encounter tasks in a different order than others. We present a (slightly edited) excerpt from a session with this system to illustrate its capabilities and limitations.

COMPUTER: On the first day of XMAS, someone's true love sent him a partridge in a pear tree. On the second day, they sent two turtle doves plus another partridge (3 gifts, 2nd day). This continued through the twelfth day, when the true love sent 12 lords, 11 ladies, ... all the way to yet another partridge. Write a program that computes and prints the number of gifts sent on that twelfth day. (This is not the same as the TOTAL number of gifts sent for all 12 days!)

STUDENT: HINT?

COMPUTER: This program requires a loop. Each execution of the loop involves accumulating the value of the index into a total.

STUDENT: HINT?

COMPUTER: Finding a total almost always means you need two things:
(1) Setting a variable to 0 before the loop.
(2) Accumulating values into that variable within the loop.

BIP has impressive features for creating flow diagrams and does an excellent job of sequencing tasks. However, its understanding of the domain is limited to a flat collection of language constructs. Based on informal analyses, BIP often rejects answers that students believe to be correct; it tries too hard to elicit a single solution, which is not always appropriate in complex domains such as programming.

BIP was hampered by its lack of understanding of planning and debugging, two central AI concerns. While BIP could individualize the sequence of tasks, it could not individualize the hints given within a task. Thus, all students who encountered the XMAS task and requested two hints would see the same two hints shown above. To improve upon BIP's pre-stored hints, our problem was twofold: to represent the underlying knowledge and to apply that knowledge in a fashion helpful to the human user.

3.2 What Human Factors Contributes

The goal of the AI specialists is to design "artificially intelligent" computer environments that tutor students in much the same way that a human teacher might tutor his students. The AI technology has progressed to the point that some very basic questions must be answered before progress can continue: What makes an intelligent human tutor successful? What are his techniques for diagnosing student problems and misconceptions? What are his

techniques for advising students? In short, how does he use his intelligence to provide tutoring superior to that provided by pre-stored hint systems like BIP? All of these questions relate to the human-computer interface, so the AI specialists at TI took the questions to the human factors group.

Job and task analyses are two of the basic tools of human factors engineering. The human factors group addressed the AI specialists' questions by setting up a system in which a computerized intelligent tutor is simulated by having an intelligent human playing the role of the computer tutor. Very simply, the human tutor observes a student's efforts by watching a monitor that is slaved to the student's work terminal. The tutor makes judgments about the student's problems and misconceptions, and types appropriate help messages that appear on the student's help terminal. It is important to recognize that, in this paradigm, the human tutor bases decisions on exactly the same information that would be available to the computer tutor, and similarly provides help the same way that the computer tutor should.

In these studies, the human tutor is carefully evaluated. Human factors specialists meticulously record all his activities, along with verbal protocols in which he explains the rationale behind his decisions. These studies are not yet complete, but a clearer model of the intelligent

human tutor is already emerging. One important trend observed thus far is that the level of sophistication required for a successful ~~human~~ tutor might not need to be as great as was originally expected.

Show Foil Number -X- Here.
(The following paragraphs, including the BASIC code.)

Here is an example of a problem a student had that was easily diagnosed by the human tutor. The student was learning how to program in BASIC, using the BIP problem set. In this particular problem, the student was asked to take two numbers, M and N, and compute their sum, difference, product, and quotient. This is what the student typed:

```
10 PRINT "WHAT IS THE FIRST NUMBER"  
20 INPUT M  
30 PRINT "WHAT IS THE SECOND NUMBER"  
40 INPUT N  
50 LET A = M + N  
60 LET B = M - N  
70 LET C = M * N  
80 LET D = M
```

At this point, the student paused for over a minute, then asked for help. Quite clearly, the student's problem was that he did not know the symbol for division. This sort of problem is representative of the type solved by the human tutor that would not have been solved by a pre-stored hint tutor like BIP. Note that even a very simple means-ends analysis model involving sequential accomplishment of subgoals is adequate to provide a correct hint here.

The student was asked to write a BASIC program that would take two numbers, M and N, and compute their sum, difference, product, and quotient. Here is what he did:

```
10 PRINT "WHAT IS THE FIRST NUMBER"  
20 INPUT M  
30 PRINT "WHAT IS THE SECOND NUMBER"  
40 INPUT N  
50 LET A = M + N  
60 LET B = M - N  
70 LET C = M * N  
80 LET D = M
```

When he got to this point, the student paused for over a minute, and then asked for help. What information does he need in order to continue?


3.3 What Artificial Intelligence Contributes

The crucial contributions of AI to CAI derive from representing the underlying knowledge. In the case of programming, representing the domain knowledge requires asking such questions as, "What is it that the expert programmer knows that the novice does not?" Miller's SPADE-0 project was more an attempt to investigate and formalize this type of knowledge than to build a useful programming tutor. It represented knowledge about programming plans (i.e., procedural templates independent of the particular programming language) and debugging techniques.

SPADE-0 built upon AI work in automatic planning and debugging developed in HACKER [Sussman 1973], MYCROFT [Goldstein 1974], and NOAH [Sacerdoti 1975]. SPADE-0 could prompt the student through hierarchical planning processes, encouraging the student to postpone premature commitment to the detailed form of the code. (This AI planning technique grew out of such systems as ABSTRIPS [ref].) SPADE-0 provided a vocabulary of concepts for describing plans, bugs, and debugging techniques, and handled the routine bookkeeping tasks involved in simple program development.

Figure XX illustrates a sample interaction with SPADE-0. The key feature is the system's deeper analysis of the underlying knowledge. This is manifested by commands for editing the plan -- rather than merely the code -- of the student's program. However, the design of SPADE-0

SPADE - Ø

PLAN:	PICTURE:
PLAN(WW) ... PLAN(WELL) REPEAT 4 TIMES <WELL-SIDE> ...	

What now?

> Run WW

{ Running WW.... Done. }

What now?

> Debug WELL

Well uses a REPETITION PLAN.

The top level contains 2 design decisions. There are warnings on the code for <WELL-SIDE>.

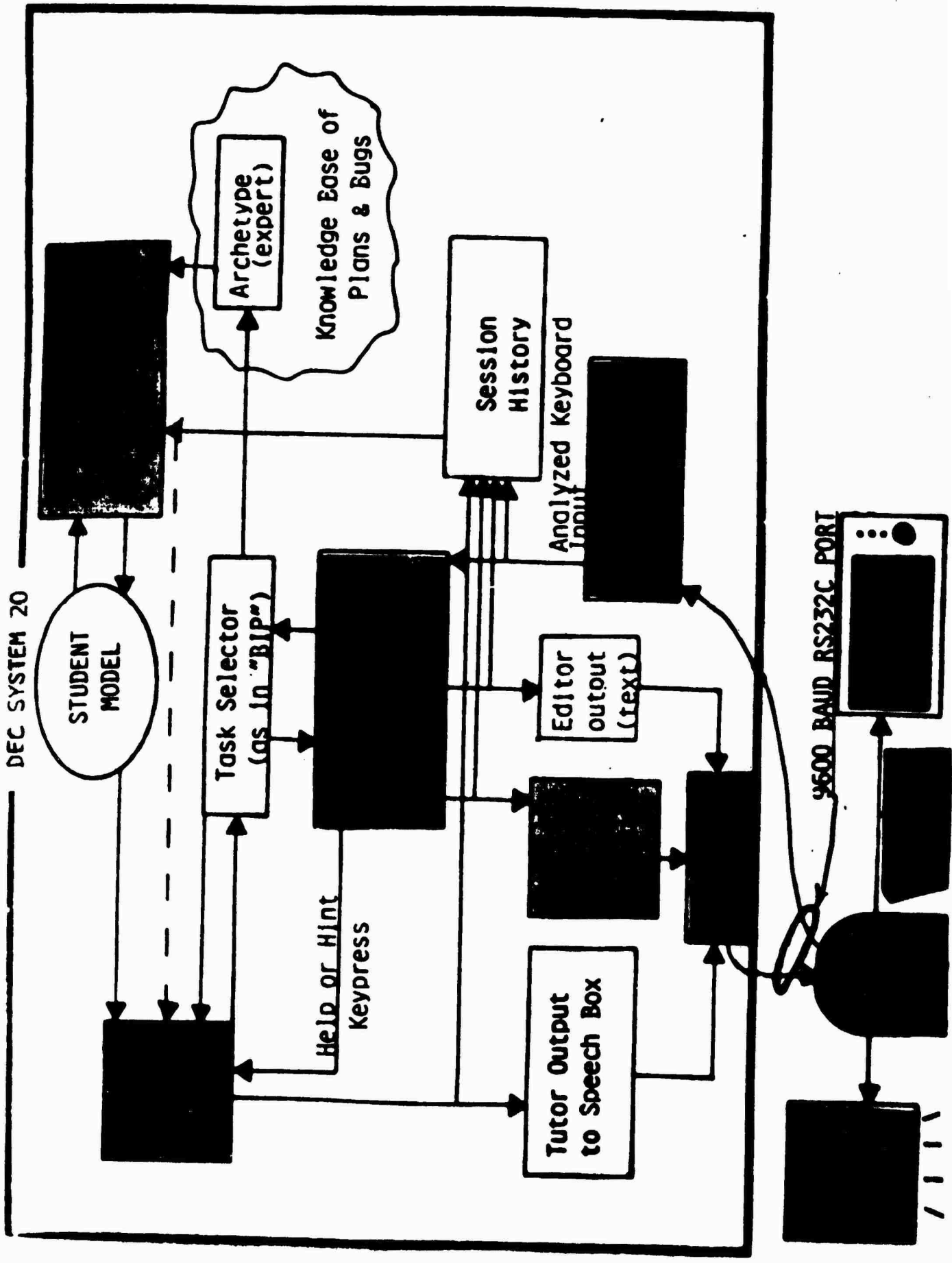
...

ignored human factors considerations, imposing its own technical vocabulary on the student, and adopting a style of interaction that took away much of the initiative.

Our current work is an attempt to extend the underlying AI knowledge represented by SPADE-0 and merge it with the improved human factors guidelines resulting from careful analyses of what good human tutors do. Like BIP, it will dynamically select tasks from a curriculum database; but like SPADE-0, it will build a model of the student's problem solving skills (rather than simply recording which programming language constructs have been mastered). The key AI aspect is fine-grained diagnosis of student errors to provide custom-generated (rather than pre-stored) advice.

We are basing the design of our new tutoring module on human factors studies in which a human simulates this module. As the system implementation progresses, additional tasks will be taken over by the computer, and the need for the human tutor to intervene will be correspondingly diminished. The proportion of tasks successfully performed by the computer tutor is a measure of our progress.

Earlier "intelligent tutoring systems" such as BIP and SPADE-0 used their intelligence to build models of the student. However, the interface between the intelligent tutor and the student remained crude. By working with human factors engineers, the AI specialists now better understand how human tutors interact with students. The emphasis of



COMPARISON OF HINTS FROM "DUMB" TUTOR AND "SMART" TUTOR

- DUMB TUTOR: "REMEMBER TO SET UP THE CORRECT HEADING AND LOCATION OF YOUR TURTLE BEFORE STARTING NEW SUBTASKS."
- SMART TUTOR: "YOU HAVE SUCCESSFULLY COMPLETED THE SUBTASKS OF DRAWING A TRIANGLE AND SQUARE. HOWEVER, YOUR TRIANGLE IS INSIDE THE SQUARE NOT ABOVE IT. CORRECTING THE INTERFACE BETWEEN THE SQUARE SUBTASK AND TRIANGLE SUBTASK WILL FIX THIS."

the AI work has now shifted to modelling this tutor/student interface.

4.8 CONCLUSION

In closing, it is worthwhile to review a central human factors problem: the division of labor between human and machine in human-machine systems. In any well-designed system, tasks are allocated to those components best suited to perform them. Textbooks on human factors engineering typically state that machines tend to be superior to humans in such tasks as calculation and coordination of many simultaneous activities. Conversely, they state that humans tend to excel in such tasks as problem solving where originality is required, pattern recognition, and decision making based on incomplete or conflicting data, or when unlikely or unexpected events occur. Thus, these guidelines would allocate responsibility for calculation to the machine, but leave the human responsible for recognizing patterns in the results of those calculations.

As artificial intelligence continues to progress, machines will begin to achieve superiority over humans in many aspects of tasks traditionally assigned to humans. This might lead to speculation that research on human-machine interfaces may be unnecessary, since the need for the human component will disappear. For certain kinds of menial tasks presently performed by humans, this line of

HUMANS ARE BETTER AT:

PATTERN RECOGNITION

**APPLYING ORIGINALITY IN SOLVING
PROBLEMS**

**MAKING DECISIONS BASED ON
INCOMPLETE OR CONFLICTING DATA**

**MAKING DECISIONS WHEN UNLIKELY
OR UNEXPECTED EVENTS OCCUR**

MACHINES ARE BETTER AT:

**ACCURATELY AND RAPIDLY PERFORMING
COMPLEX CALCULATIONS**

**COORDINATING AND PERFORMING MANY
SIMULTANEOUS ACTIVITIES**

**PERFORMING ROUTINE OR REPETITIVE
TASKS**

MONITORING

reasoning is probably sound. However, it is our expectation that, as work in artificial intelligence and human factors engineering continues to advance, the nature and power of the human-computer interface will become more critical and sophisticated. The art and science of interface design will never become obsolete. Obsolescence is faced only by our traditional task-allocation guidelines.

This paper has described two examples of research projects in which AI and human factors specialists have collaborated. From these projects and others like them, we have learned to stop thinking in terms of separate disciplines that merely benefit from cooperation. Particularly in the design of "intelligent interactive systems," the borderline between these two fields has blurred in our eyes. Human factors specialists are learning to exploit the tremendous benefits for the human component made possible by more intelligent software components; AI specialists are learning to write software that is sensitive to the needs, capacities, and limitations of the human component. Due to this kind of synergism, the well-designed human-computer interface can become a link between the creative thoughts of men and machines, contributing to a technological revolution that offers to do for the human mind what the industrial revolution did for human muscle.

5.0 REFERENCES

- Barr, Avron, Marian Beard and Richard Atkinson. The Computer as a Tutorial Laboratory: the Stanford BIP Project. International Journal of Man-Machine Studies, 8, 1976, pp. 567-596.
- Burton, R.R. Semantic Grammar: An Engineering Technique for Constructing Natural Language Understanding Systems, BBN Report No. 3453, 1976.
- Chapanis, A. Man-Machine Engineering. Belmont, California: Brooks/Cole, 1965.
- Chapanis, A. Interactive human communication. Scientific American, 1975, 232(3), 36-42.
- Ford, W. R., Weeks, G.D., & Chapanis, A. The effect of self-imposed brevity on the structure of dyadic communication. The Journal of Psychology, 1980, 104, 87-103.
- Goldstein, Ira. Understanding Simple Picture Programs. Massachusetts Institute of Technology Artificial Intelligence Laboratory, Technical Report ???, 1974.
- McCarthy, John, et al.. LISP 1.5 Programmer's Manual. MIT Press, 1965-6.
- Michaelis, P.R. Cooperative problem solving by like- and mixed-sex teams in a teletypewriter mode with unlimited, self-limited, introduced and anonymous conditions. JSAS Catalog of Selected Documents in Psychology, 1980, 10, 35-36 (Ms. No. 2066).
- Michaelis, P.R., Chapanis, A., Weeks, G.D., and Kelly, M. J. Word usage in interactive dialog with restricted and unrestricted vocabularies. IEEE Transactions on Professional Communication, 1977, PC-20, 214-221.
- Miller, Mark. "A Structured Planning and Debugging Environment for Elementary Programming." International Journal of Man-Machine Studies, January 1979.

Minsky, Marvin. "Matter, Mind, and Models." Proceedings of International Federation of Information Processing, 1966.

Newell, A., & Simon, H.A. Human Problem Solving. Prentiss Hall, 1972.

Papert, Seymour. Mindstorms. Basic Books, 1980.

Sacerdoti, Earl. A Structure for Plans and Behavior. Publisher???, 1975.

Schank, R.C. Conceptual Information Processing. New York: Elsevier, 1975.

Sussman, Gerald, A Computational Model of Skill Acquisition. Massachusetts Institute of Technology Artificial Intelligence Laboratory, Technical Report 297, August 1973.

OVERVIEW OF SELECTED DISPLAY FORMATTING AND CLUTTER REDUCTION TECHNIQUES^{1,2}

Franklin L. Moses
Human Factors Technical Area

US Army Research Institute for the Behavioral and Social Sciences
Alexandria, VA

System and software designers for graphic applications have a real dilemma. Designers often are given the type of symbols to be displayed, the amount of information to be portrayed, and the hardware to be used. If they cannot change the symbols, reduce the data, or replace the hardware, what can be done to make a display speak to the user with the clarity desired? One solution is to format the information so that the display is compatible with the user's perceptual abilities and task requirements. The essence of such formats is to highlight information relevant to a task and thereby make it stand out from the irrelevant information.

The goal of creating "good" displays is to present information so that user needs can be satisfied quickly and efficiently. However, one problem created by adding more information to a display screen, even if it is relevant to the user, is generally called clutter. For the sake of discussion, clutter exists when the extraction of information from a display is hindered by the density or similarity of symbols. A number of alternative formatting techniques can be suggested to reduce clutter. Of course, some methods will work better than others, depending on the situation.

Although the examples of formatting in this paper all relate to Army applications, the principles should easily generalize. Army representations of the battlefield illustrate a classic problem for displays: or users try to display more information, they end up extracting less due to clutter. Formatting guidelines are needed to help reduce the clutter problem.

Formatting Situation Displays

Figure 1 is a typical, albeit fictitious, Army battlefield map. Anyone who has seen a real one will recognize this one as a severely stripped down version. It shows only the most essential information: terrain (mountains, rivers, roads and forests); the unit type (artillery, infantry, armor); and the unit sizes (division, brigade, and battalion). Yet, it already is cluttered. Consider the time and effort that a person would need to compare the number of armor, artillery, and infantry units, even on such a simplified display. Alternative formats using the same symbols and the same information can help to make such tasks easier for the user. Several suggestions, based on Army Research Institute (ARI) work, should allow more information to be meaningfully displayed without adding hardware costs or decreasing user performance.

¹An earlier paper by Leon H. Gellman (currently at Sarah Lawrence College, N.Y.) was presented at the US Army Second Computer Graphics Workshop, Virginia Beach, VA, September 1979, and used as a basis for the current report.

²The views expressed by the author do not necessarily reflect the views of the US Army or of the US Department of Defense.

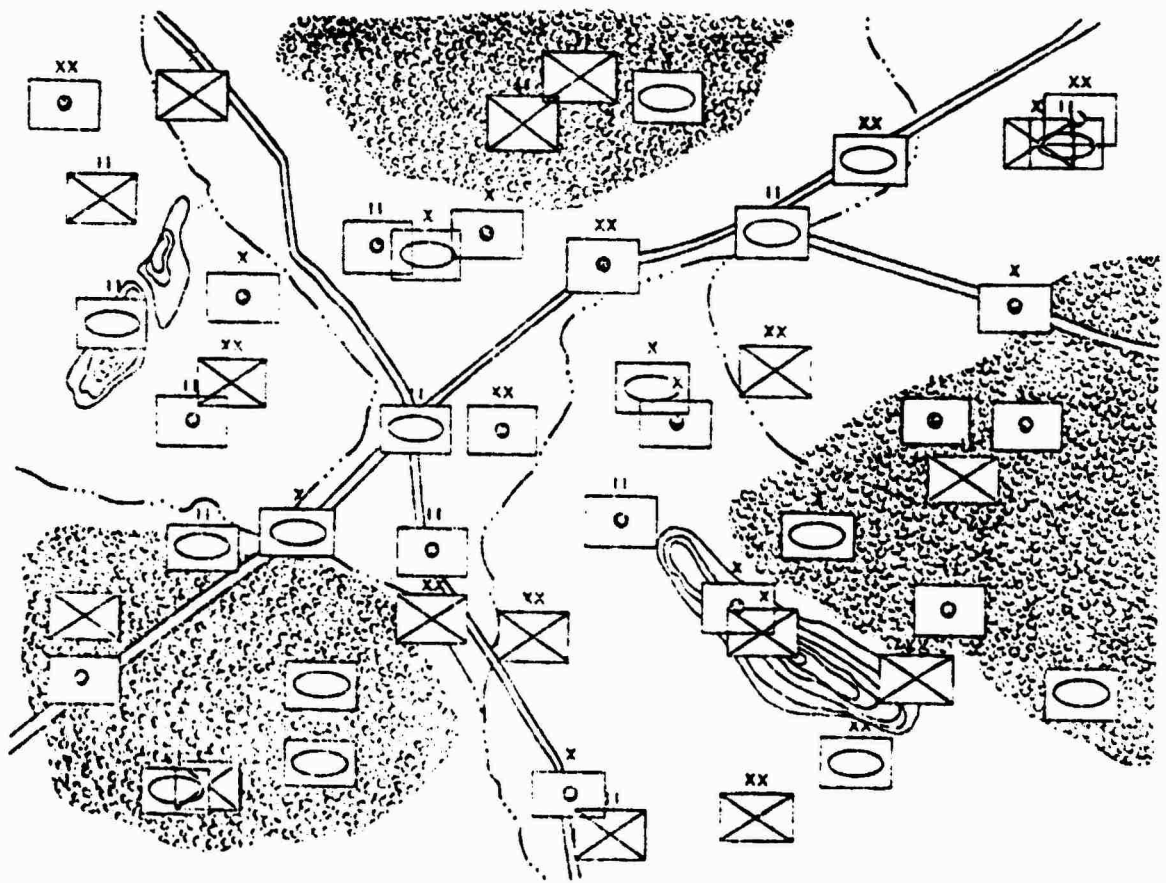


FIG. 1

Redundant Codes

The first formatting technique to be discussed is based on the research of Vicino, Andrews and Ringel (1965). They doubly or redundantly coded information on a battlefield display, thereby allowing users two chances to find the information. Redundant coding takes information which is already on the display and repeats it in a salient code that helps the user to organize the display. For example, Figure 2 presents the map with redundantly coded unit symbols. The code is the heavy broken line for artillery, the heavy rectangle for armor and the heavy X for infantry. There is no more or less information here; rather, there are two ways of identifying the units. The double code has been used to maximize the saliency of unit types making similar units seem to stand out together. When Vicino et al. used this technique, they increased the speed of information extraction by 97% when compared with a single code. Redundant codes will not necessarily increase processing speed this much in all situations. However, processing should be easier and the cost of such formatting is minimal. Redundant coding can be done with any number of stimulus dimensions such as blinking, size, intensity and color.

Sequential Formats

Sequential Presentation by Topographic Segments. So far, the discussion has centered on using codes to organize display content. If a display has to show a lot of detail, then a second type of format, called sequential presentation, organizes the information by breaking it up into component parts. This is accomplished by showing information in segments over time. Sequential presentation reduces clutter by showing less information per screen and, for similar reasons, it increases the amount of detail that users can see. The technique is particularly useful for showing standard topographic information that easily exceeds state-of-the-art display resolution capabilities.

Sequential formats require users to depend on their ability to integrate information over time. Thus, an important formatting question concerns whether to display segments of an entire map by scanning them or by sequentially presenting static (i.e., discrete) views. Based on an ARI experiment by Moses and Maisano (1979), static views with overlaps of around 25% are more efficient for users than continuous scanning methods of sequential map presentations. When resolution and clutter are serious problems, sequential presentation should be considered as a solution.

Sequential Presentation by Data Dimension. The final formatting technique to be discussed is also a sequential presentation method, but this one displays information by data dimensions. The idea is once again to segment information. This is accomplished by presenting a limited number of data dimensions simultaneously while removing other information from the screen. Of course, questions such as how many separate data dimensions can be shown per screen and what is the effect of user control over selection of dimensions need to be considered. These and other inquiries about sequential presentation are topics for possible future investigation at ARI.

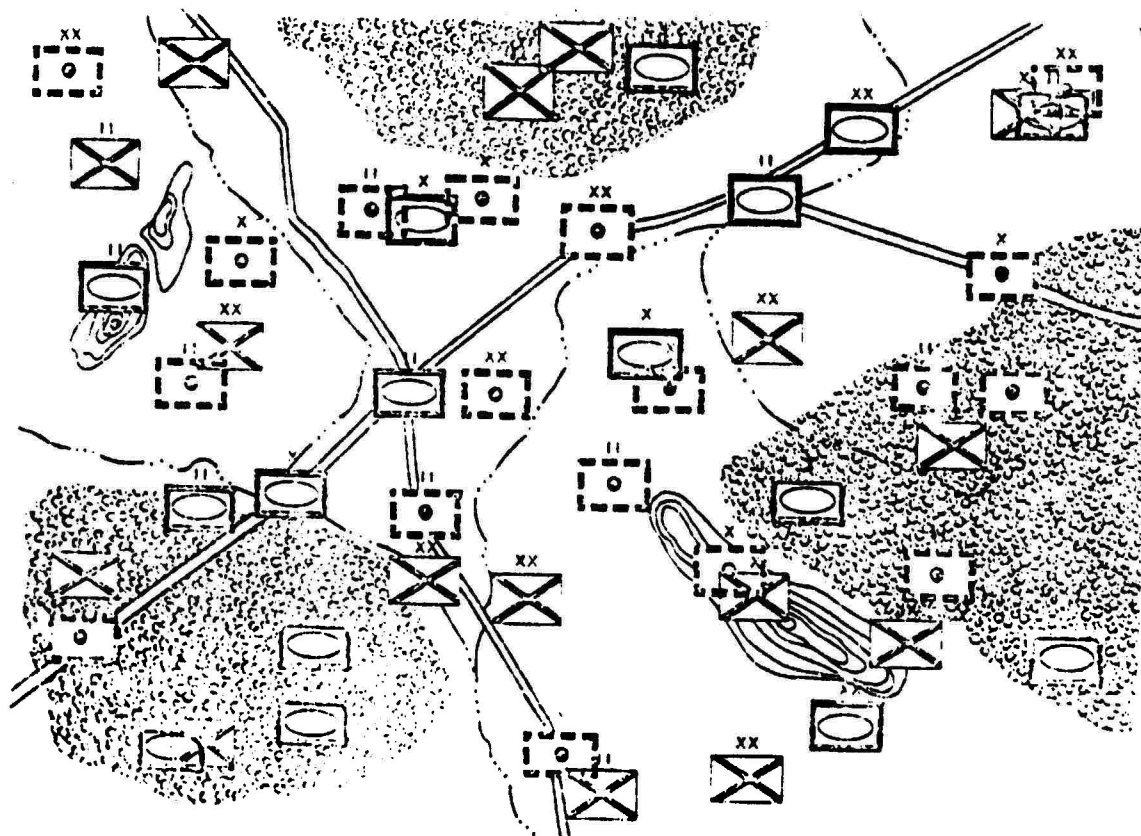


FIG. 2

Summary

This paper discusses the problem of putting too much information on a display and outlines four formatting techniques which may alleviate some effects of clutter. The suggested formatting techniques are only a few of many methods available to the graphic system designer. The question that remains is: Which format should be used? The answer can only be found by determining the format that optimizes task performance for display users. Clearly, none of the recommendations made here will provide an unconditional solution to graphic problems. However, it is incumbent upon the designer and programmer to use every trick at their disposal to provide graphics which have the impact and clarity commonly believed possible. The Workshop presentation will consider this goal in more detail along with some guidelines for attaining it.

References

- Moses, F.L. and Maisano, R.E. User Performance Under Several Automated Approaches to Changing Displayed Maps. ARI Technical Paper 366, June 1979.
- Vicino, F.I., Andrews, R.S. and Ringel, S. Conspicuity Coding of Updated Symbolic Information. APRO (now ARI) Technical Research Note 152, May 1965.

FORMAL GRAMMAR REPRESENTATION OF MAN-MACHINE INTERACTION

Phyllis Reisner
IBM Research
5600 Cottle Rd.
San Jose, CA 95193

End users communicate with a computer system by using a language. The language might be, for example, a query language, a natural language, or an "action language" - a sequence of button presses, typing actions, cursor or lightpen actions, etc. These user input languages can be represented in the same way as any other language - by a formal grammar which shows the permitted strings and also shows the structure of the language.

The work to be described in this talk attempts to use a formal description of the user input language as a design tool to improve the ease-of-use of a man-machine interface. The talk will first describe earlier work, which uses a BNF-like grammar in the context of a color-graphics system for making slides. It will then discuss current work using a formal grammar to describe text editing. The current work is first attempting to make some of the concepts introduced informally in the earlier work sufficiently precise that people with a variety of backgrounds can use them.

The field of human factors, which attempts to measure and improve the ease-of-use of products, is largely experimental. It uses techniques of behavioral science as its primary methodology. The intent of the work with the color-graphics system was to demonstrate that a formalism could

be applied in this area which is usually considered soft, or even ad hoc. The intent was also to explore the possibility of using the formalism to compare alternative designs for ease-of-use and to located design flaws that might cause user problems. We wanted to see whether a tool could be developed that had some predictive potential. One problem with the usual behavioral approach to interface design is that it must frequently await the existence of a prototype or working model. We wanted to augment this approach with a more analytic one.

The color-graphics system, ROBART, existed in two versions, ROBART 1, which was designed without explicit attention being paid to ease-of-use, and ROBART 2, a redesigned version with the end-user a major focus of attention. It was an experimental, interactive system for creating slides for technical presentations. It was intended to be used by people without computer training doing non-routine tasks. The function available in both versions was essentially the same, but the design of the human interface differed.

To explore the issues discussed, the "action language" of the first version was described, using a BNF-like notation. (In this action language, the user selected colors by dipping a cursor into a paintbox of colors on a CRT screen by using a joystick, selected shapes such as lines, circles, rectangles, etc. by various combinations of switch selections and button presses on an external switchbox, indicated the location and orientation of the shapes by combinations of cursor positioning and button presses. It was also possible to type textual material on the screen, in color). Portions of the action language for ROBART 2 were also described,

also using the BNF-like notation.

The next step was to make predictions, from these formal descriptions, about very specific differences in the ease-of-use of the two versions, and then to test the predictions to see if they were in fact substantiated. The goal was to see if formal grammar could be used as a predictive tool and if the predicted differences were indeed measurable.

This did indeed turn out to be the case. Among others, we predicted that the action of selecting shapes would be more difficult in ROBART 1 than in ROBART 2, for each of the shapes available. We also predicted that users would make a particular error in "initiating" shapes (the first action to indicate location and orientation) in ROBART 1 and would not make an error in the same step for ROBART 2. Since the same error was not expected to occur in ROBART 2, we felt that the problem would indeed be attributable to the interface design and was not inherent in the function itself.

In an exploratory experiment with temporary office workers, the predictions were in fact substantiated.

Current work, in the context of text editing, is first attempting to clarify some of the concepts and techniques used in the above work. The concepts were intuitive, but not precise enough to develop into a design tool to be used by a variety of people with different backgrounds. For example, we introduced the notion of a "cognitive" terminal symbol, since we thought that what the naive user has to learn and remember will be of major importance in the ease-of-use of a system he uses intermittently.

This notion clearly needs to be made more precise. We also used a quasi-automatable technique for locating structural inconsistencies in the language. We expected these structural inconsistencies to cause users to make mistakes. Neither the notion of "structural inconsistency" nor the technique have been made explicit. These and other related issues will be discussed.

A RULE BASED HELP SYSTEM FOR SCRIBE

ELAINE RICH

AARON TENIN

26 February 1961

People need access to help if they are going to use complex computer systems effectively. There will not always be other people or even manuals around to help them. So we need the computer itself to be able to provide the help its users need. This is not a new argument. See, for example, [Pirtle 68].

The extent to which anyone can help someone else is limited by the depth of the helper's own knowledge. So if computers are going to help people, they must have a great deal of knowledge about what they do.

But the usefulness of help information to a person seeking help is a direct function of the extent to which the information answers the specific question the user had. So simply dumping an entire manual or even large chunks of it on a user every time he asks a question is useless.

People who need help are missing some information about how the system works. So they cannot be counted on to describe their problem in terms of specific system commands so that the relevant parts of the manual can be found and fed back to them. (This precludes simple keyword based help systems such as [Shapiro 75] or [Kenler 80].)

These obvious facts force us to the conclusion that to provide a good interactive help facility will require a large data base of knowledge about the operation of the system in question. This data base must be structured in such a way that it can be accessed from descriptions at a variety of levels about what the program did and what the user wanted. To investigate the issues

raised by such constraints, we are building a help system for the document formatting program Scribe [Reid 80].

The knowledge base used by the system is a set of rules that describe Scribe's behavior at a variety of levels. Top level rules describe the behavior of the system in terms of fairly high level functions. Other rules then describe those functions in terms of lower level functions, and so forth. We plan initially not to try to provide rules that describe Scribe down to the lowest level, at which individual characters are placed on the page. This will of course limit the ability of the system to answer questions about that aspect of the system's performance. But this is analogous to the situation that occurs with human consultants. There comes a point where, unless they are familiar with the details of the code of the system, they simply cannot answer a question. This rule based, successive decomposition approach, however, prevents us from being locked into a particular level of description. New rules that provide additional levels of description can be added at any time.

Each rule in the system contains a left side that describes when it can be invoked, and a right side that describes the sequence of actions that will result. The left side consists of two parts, a command or a piece of the input file, which tries to trigger the rule, and a list of auxiliary conditions that must be met in order for the rule to be able to be invoked. For example, the following rules describe how Scribe processes the @ref(arg) command, which substitutes for the string "@ref(arg)", the reference indicated by the string arg. (Commands to Scribe are

signalled by the character "@",.):

- 1 @ref(arg) and lookupstymboltable(arg) NEQ 0 ->
 send(maintext, lookupstymboltable(arg))
- 2 @ref(arg) and lookupauxfile(arg) NEQ 0 ->
 send(maintext,lookupauxfile(arg))
- 3 @ref(arg)->
 send(maintext,@c(arg))
 send(errorfile,"undefined reference",arg)

The order of the rules in the data base reflects the order in which Scribe checks for things. In this example, Rule 1 says that if there is a ref command with a particular argument and if there is an entry in the internal symbol table indicating a previous definition of that argument, then print in the output the appropriate value as indicated by the definition. Otherwise, if there is a definition of the argument in the AUX file (a file containing the symbol table that was built the last time Scribe processed this file) then use that definition. If there was no definition in either place, then simply insert into the text the string that was the argument to ref, but capitalize it. Also make a note of this error in the error log file.

The actions indicated by these rules are fairly high-level. They indicate that text should be placed in output files. They do not indicate how. They do not specify such things as the margins or the type font to be used. Those things are specified in the rules that describe the operation of the send function. Some of the actions, such as send, can only be generated by the operation of other rules. Others, such as @c(text), could also have occurred in the input file. The fact that the Scribe system

is very well structured makes it easy to describe the operation of one function in terms of a well defined set of other functions. This one-step-at-a-time description is very important for the generation of responses to user's questions. No one wants a bit level answer to every question they ask. People usually want a description in terms one or perhaps two levels higher or lower than the level at which they asked the question.

The set of rules provides a static description of the way operations in Scribe are performed in terms of other, lower level operations. As Scribe executes, it builds a separate hierarchical structure that reflects the block structure of the specific document that is being processed. For example, a document could contain the sequence:

```
@begin(quotation)
  @begin(itemize)
  @end(itemize)
@end(quotation)
```

The quotation environment specifies that the margin should be moved in and that the text should be printed single spaced. The itemize environment specifies that the margins should be moved in and that paragraphs should be numbered. These specifications nest, so that the margins inside the itemize will be narrower than for the rest of the quotation, which will be narrower than the surrounding text.

To answer a user's questions, the help system will match pieces of the user's question against pieces of rules, and use unmatched

pieces of the rules or patterns of chaining through the rules as answers to the questions. Many questions can be answered by referring only to the static description of Scribe's operation. However, when a question refers to something specific that happened at a particular point in the user's file, it may be necessary for the help system to build a piece of the dynamic tree, mirroring that built by Scribe during execution, so that it will know enough context to be able to identify the rules that were applied.

One of the most common types of questions a help system must answer is "why did X occur?". This usually means that the user expected that something else would occur. To answer such questions, the help system finds the rules whose right hand sides specify the effect the user has described. Let's assume, for simplicity, that there is exactly one such rule. Now a superficial answer to the question is simply to state the left side of that rule. But much of what is there is usually redundant. For example, the user knows what command he specified. What the help system will do is to compare the rule it found to others whose left sides are different. The differences in the left sides are the specific reasons why the observed effect occurred, rather than some other. So, for example, if the user asks why his `eref` command resulted in the label and not the thing to which it referred being printed, the system observes that this happened because the label was not previously defined. It concluded this by comparing Rule 3, the one that describes what Scribe did, to Rules 1 and 2, which

describe what it would have done if things had been slightly different.

Sometimes there may be a great many rules whose left sides almost match the selected rule. It may then be necessary for the helper to ask the user what he expected to have happen. Then only the rules whose right sides match that expected action need to be considered. Ideally the system would maintain a good model of the user so that such questions would rarely need to be asked. Sometimes general knowledge about the way people use the system will help here. For example, people usually expect some fairly direct connection between the commands they issue and the results they see. They rarely expect a command to be a no-op. But there will always be times when an individual has an idiosyncratic misunderstanding of the system and nothing short of a direct question will point this out. For this reason, the process of answering a question must be thought of as a dialogue rather than as a one-shot question and answer.

Another common type of question is what Genesereth (Genesereth 74) calls the "howdo" question. For example, "How do I get my footnotes to come out at the end of my document rather than at the end of each page?". Howdo questions are answered by matching the user's description of what he wants to do against the right sides of the rules to find those that can produce the desired effect. If there are more than one, then the choice among them will be made by considering such things as the complexity of the constructs involved and the user's level of expertise with the system. The left side of the chosen rule describes what is

necessary to accomplish the desired effect. But it may contain conditions that the user cannot specify directly. So the help system must chain backwards through the rules to find the commands that will cause those conditions to be true.

Yet another common type of inquiry is the "what is the difference between" question. For example, a Scribe user might ask, "what is the difference between the itemize and enumerate commands?". These questions can be answered easily by this kind of rule based system without having been anticipated in advance. It need merely find the rules that describe the operation of each command by matching against left sides. In the simple case, there will be one rule for each and the answer to the question is simply a list of the differences between the corresponding right hand sides. In more complex cases, it will be necessary to compare left hand sides also to determine the effect of various other factors on the operations of the two commands.

One of the most common situations in which users ask questions is when they have gotten some kind of error message from the system. Talking about such errors is easy for a rule based system. The rules describe all the things the system can do and the situations in which it will do them. Errors do not need to be represented explicitly. They are implied by the absence of rules. If the user wrote a command X and there are no rules for command X whose other preconditions were satisfied at the time the command occurred then an error will arise. The system can explain the error by comparing the existing state to the required preconditions and reporting the differences. This is extremely

useful, since for a complex system the number of possible error configurations can be very large and it would be very difficult to have to describe each of them explicitly.

A good help system must tailor its responses to the needs of individual users. In this it is no different from other interactive systems [Rich 79]. One way to represent a model of a Scribe user would be as a set of rules, presumably a subset, possibly with errors, of the rules that the system knows. With such a model, some question would be very easy to answer. For example, why questions could be answered by comparing the user's rules against the system's correct rules to find the difference and report it. This technique was suggested by Burton and Brown [Burton 76] as a way an intelligent CAI system could discover bugs in a student's knowledge. But it is unreasonable for a help system to maintain such a massive amount of information about each user. Instead, we propose to record a very small number of facts about each user, such as a measure of his expertise with the system. Each of the objects used in the system will have associated with it some properties, some of which can be matched against user characteristics to determine the appropriate rules to use in generating a response to the question. So, for example, commands will be marked as simple, intermediate, or advanced. Other factors that should be included in the model of each user are his inclination toward being a hacker (i.e. does he want to learn fancy new commands or does he want to know a way to get by with the commands he knows?) and his familiarity with computer science concepts (such as block structure, one pass

system, symbol tables).

One of the major advantages of this rule based representation of the knowledge required by an intelligent helper is that it mirrors the structure of the system for which the help is being provided. (Or at least it does if the system is well structured.) This suggests that the top down process of writing the rules could be used to produce a well structured program and its help system simultaneously. We would like eventually to try to build an entire system this way.

REFERENCES

- [Burton 76] Burton, Richard & John Seely Brown.
A Tutoring and Student Modelling Paradigm for
Gaming Environments .
In *Proc. of the Symposium on Computer Science and
Education*. 1976.
- [Genesereth 78] Genesereth, Michael.
Automated Consultation for Complex Computer
Systems.
PhD thesis, Harvard, 1978.
- [Kehler 80] Kehler, T. P. & M. Barnes.
Alternatives for On-line Help Systems.
In *Proc. 11th ACM SIGUCC User Services Conference*.
1980.
- [Pirtle 68] Pirtle, Melvin.
Help.
In *Conversational Computers*, . John Wiley & Sons,
New York, 1968.
- [Reid 80] Reid, Brian.
Scribe: A Document Specification Language and its
Compiler.
PhD thesis, Carnegie-Mellon, 1980.
- [Rich 79] Rich, Elaine.
User Modeling via Stereotypes.
Cognitive Science :329-354, 1979.
- [Shapiro 75] Shapiro, Stuart & Stanley Kwasny.
Interactive Consulting via Natural Language.
Communications of the ACM :459-463, August, 1975.

Models for the Design of Static, Software Systems

M.L. Schneider
Sperry Univac
Blue Bell, Pa 19424

1. INTRODUCTION

One of the "axioms" for ease-of-use is: "Help systems are necessary" (Clark 1980). While an increasing number of software systems provide some form of user assistance (Relles 1979), the information is usually provided without regard to its useage. In general, assistance is nothing more than an "electric reference manual."

When factoring exists, it usually consists of a layered approach; the user can request additional details about a specific topic. This addresses the problem of verbosity, but only indirectly considers the expertise level of the requester.

This paper proposes cognitive factors that may impact information factoring: different levels of user sophistication (the User Taxonomy) and different segments of task performance (the Transaction Taxonomy). The interaction between these two taxonomies can provide guidelines for improved static information factoring in assistance systems.

2. USER SOPHISTICATION TAXONOMY

The developmental levels of computer language acquisition defined in this taxonomy are

1. Parrot
2. Novice
3. Intermediate
4. Advanced
5. Expert

Each level is characterized by skills in language production: item, field, or statement chunking; breadth of language scope; and degree of generalization or abstraction of concepts. The change in system knowledge is manifested through an increased competence in the commands that are regularly used and an awareness of additional functions available within the system or language.

The basis for this taxonomy arises from qualitative observations of computer usage in a wide variety of software systems and the relationship between the observed computer productions to those observed in the natural language development. This taxonomy describes an individual's expertise or sophistication in a single software system or language (or subset thereof) and may not be transferable. The level at which an individual stops progressing appears to depend upon a number of factors related to the learning of complex tasks and the demands placed upon the person by the task requirements.

2.1. THE PARROT

An individual at the lowest level in the taxonomy, the Parrot, has minimal knowledge of the computer system. The Parrot approaches the computer system and types commands. This individual does not think, question, understand, or synthesize the commands. These commands, or sequence of commands in some cases, may be moderately complex. Satisfaction is derived simply by having the computer perform the task.

When the question "What am I doing?" is asked, the Parrot is ready to progress to the next stage of sophistication: the Novice.

2.2. THE NOVICE

With experience, a user begins to understand several isolated concepts and is able to choose a specific lexical entry (command) for a function. The user is required to know specific but not complex information. Semantically, the items are considered in the concrete, not in the abstract. The Novice may ask, "What does this command item do?" not "What can it do?" By now, the user has a minimal command of the grammar, but is only able to operate on an item-by-item basis. For example, the Novice may recognize a verb and one or more objects in a command, even if the grammar allows modifiers in the verb phrase or in the object phrase.

Unlike the Parrot, the Novice analyzes each item, thus extracting lexical information. The language components now have meaning and can be used in a flexible manner.

2.3. THE INTERMEDIATE

The Intermediate is a level between the Novice and the Advanced user. Whereas the Novice concentrates on items in isolation, the Intermediate operates with items in fields and with fields in statements. A statement now becomes the primitive conceptual unit. The use of a larger chunk encourages syntactic and semantic conciseness in the grammar, allowing the user to minimize keystrokes.

At times, the Intermediate user may link statements into command "chains" such as compile...collect...execute. Even so, each command is still considered in isolation. The user generally waits until a function has been completed before proceeding to the next request, wishing to see the result of a command before continuing with the task.

The Intermediate begins to concentrate on the task rather than its components. Use of the full language may be restricted by a lack of knowledge. Thus, the Intermediate continues to expend significant effort on language details. At this point in the user's development, the more subtle grammatical rules become evident. A Novice would use a default, unaware of the fact that an item can be specified. An Intermediate would consciously use a default in order to reduce keystrokes or save time. Initially the Intermediate uses knowledge in a specific problem domain. Later, this information is generalized, allowing new problems to be solved.

Toward the end of the Intermediate level, considerable skill in the understanding and manipulation of a segment of the command set has been achieved. With the increased use of larger syntactic chunks each requires less attention. This is the process of automatization. Thus, increased attention can be given to the entire task, rather than to the mechanisms required for its performance.

With further experience and increased task requirements, the Intermediate can evolve into an Advanced user, subordinating the computer language to the task.

2.4. THE ADVANCED USER

Whereas the Intermediate attempts to solve problems via a series of isolated commands, the Advanced user realizes that an interconnected collection of statements can be more productive for certain tasks. At this level a program or procedure, rather than a single statement, results. Because commands are now interrelated, the scope of the syntax and semantics expands. The syntactic elements are abstract rather than concrete. Data structures provide the vehicle for producing abstract objects. For example, a variable would be used to represent a filename or a string. The Advanced user continues to retain the command, together with other defined procedures, as language primitives.

Control structures are useful if the direction of flow between statements is to be modified. Using these structures requires a modification of the user's thought process. A Novice or Intermediate user may not foresee the success or failure status of a command as an object on which operations are defined. An Advanced user thinks about the possible outcomes of commands and has the ability to take appropriate action. While Novice and Intermediate users operate with concrete syntactic constructions, existing within a specific, restricted semantic scope, the Advanced user expands his language knowledge to cope with complex structures and abstractions.

Practically speaking, the Advanced user has the ability (though not necessarily the need) to accomplish any function within the system. The Advanced user is completely facile with the language and can deal with the language at the global "metalinguistic" level.

2.5. THE EXPERT

The Advanced user has the ability to use the language with relative ease. Since any computer language is restricted in scope, it can limit a user (for example, the inability to have abstract data types in FORTRAN 77). The Advanced user, knowing the scope of the language, is constrained when faced with a new problem whose solution cannot be derived from existing functions or objects within the system. The Expert transforms this finite system into a generative one. When faced with the above situation, he creates, not derives, a new syntactic element within the system. Thus the Expert expands the existing system, creating new objects and functions.

3. TRANSACTION TAXONOMY

While the sophistication level of the user is important, it is necessary to know how a transaction is processed in order to acquire additional assistance information. A transaction is defined as the task contemplated by the user (For example: writing a program, "checking-in" an airline passenger, or performing a data base query).

The five stage transaction taxonomy shown below builds upon a simple taxonomy (command and data input, processing, and system output) by expanding the first operation, input, into its semantic and syntactic components as suggested by Shneiderman (Shneiderman 1979).

STAGE	ACTION
I	Task Analysis
II	Semantic Analysis
III	Syntactic Analysis
IV	System Performance
V	Response Analysis

3.1. STAGE I -- TASK ANALYSIS

In the first stage the user decomposes a single conceptual task into its component subtasks and determines the specific commands required for task completion. The user asks the question, "What steps and commands are necessary to perform the overall task?" For example, running a program (the single conceptual task) may require the following subtasks: editing, compilation, collection, and execution. It is possible that more than one step can be included within a single command (for example a compile-load-go) or more than one subtask is required within each subtask (for example operations with the editor).

The cognitive processes at this stage may include all or some of the following steps:

1. Identification of the full task.
2. Decomposition of the task into its subtasks or steps.
3. Definition of the conceptual operation for each step.
4. Choice of the appropriate command for the implementation of each step.

It should not be assumed that all commands will be chosen at the outset. It is highly probable that an individual will determine the conceptual operation for the first subproblem, choose an appropriate command, perform it, assess the result, then progress to the next conceptual operation, the choice of which may be influenced by the result of a previous task.

Once the conceptual operation has been defined, a user may wish to examine the set of commands for its implementation. It is possible to relate commands and conceptual operations in two ways: define a conceptual operation for commands that are conceptually related, or its antithesis, to extract from a conceptual operation its constituent commands. By iterating between these perspectives, it should be possible for the user to determine a command that allows the conceptual operation to be performed.

A command subset of a hypothetical editor illustrates this iterative approach. Consider the command "LOCATE" (this searches the text printing the lines whenever a string occurs). The specific to general relationship would be:

"LOCATE" ----->search
 print

The general concept print may refer to a number of commands that, if successfully executed, print a line:

print ----->"PRINT"
 "LOCATE"
 "FIND"
 "GOTO"
 "NEXT"

If all commands of the concept search print a line, then the structure could be represented as:

print ----->"PRINT"
 "GOTO"
 "NEXT"
 search ----->"LOCATE"
 "FIND"

A similar grouping can occur for "GOTO" and "NEXT".

When explanations are provided (basic semantic information) within the above framework, the user can obtain the information in a unified manner.

3.2. STAGE II -- SEMANTIC ANALYSIS

In the second stage, the scope of the command is considered by the user. Upon entry to the semantic analysis, the command is conceptual in the broadest sense. Now it must be refined into its detailed semantic components.

The question: "What do I want to do?" is asked by the user. The user must be cognizant of two semantic concepts: definition of the data and the control of the process. A sorting program illustrates the type of information considered by the user. A user must be aware of the data restrictions (eg. numerics only, alphanumerics, maximum number of items, maximum number of fields, etc.) and the method(s) of data storage or entry. In addition, information is required to control the processing (ascending, descending, key(s), collating sequence, etc.). At the semantic stage, it is unnecessary to know how to encode this information.

3.3. STAGE III -- SYNTACTIC ANALYSIS

When a user reaches the third stage, encoding the information, the correct function has been chosen and the semantics for task completion are understood. Now the question is, "How do I do it?" The translation of the conceptual operation into the input format is purely mechanical. The user requires syntactic information and techniques that facilitate this transformation. The form of the human-computer interface (command language, dialogues, menus, function keys, etc.) has a primary impact at this stage.

3.4. STAGE IV -- SYSTEM PERFORMANCE

System response, the fourth stage, can be treated as a "black box". The underlying architecture that supports the interface is outside the scope of this paper.

3.5. STAGE V -- RESPONSE ANALYSIS

The analysis and interpretation of the response produced by the software is the final stage of a transaction. The user now asks, "What have I done?" The primary goal of a response is to provide the user with relevant information. Unnecessary details that obscure this information should be avoided. Two independent topics should be considered: verbosity and information content (Schneider 1990).

For example, if the task is to assign the file, MYFILE, there are a number of possible responses if it is successful (ordered by increasing verbosity and content):

1. > {a prompt for the next command}
2. READY {, OK, COMPLETE,...}
3. File MYFILE has been assigned.
4. MYFILE assigned with the PUBLIC, and CATALOG options.
5. File MYFILE has been assigned. It can be used by anyone (PUBLIC) and will exist for one day (CATALOGUED) unless otherwise requested. To keep the file longer than one day contact the file administrator.

The last response is an example of layering. Three items of information have been displayed:

1. The name of the assigned file
2. The file attributes
3. The administrative procedure required to keep the file.

In a similar manner, it is possible to design a layered HELP function (a user initiated request for assistance).

A command may not always terminate successfully. Useful and meaningful error messages are important. Good error reporting should provide sufficient information for the user to:

1. Understand the nature of the error;
2. Understand the source of the error;
3. Understand the methods for recovery or correction.

Again the questions of verbosity and information content are important. Verbosity may be correlated with the number of times an individual has seen the message, while information content should be related to the levels in the user taxonomy and task requirements.

4. INTERACTIONS BETWEEN TAXONOMIES

The user and transaction taxonomies should not be considered in isolation. Based upon the sophistication level of the user, the scope of assistance may vary. Different segments of the transaction taxonomy need to be emphasized or deemphasized. The method of assistance presentation provided to individuals at different sophistication levels for the same transaction may differ. For example:

C: FILE MYFILE HAS BEEN ASSIGNED
u: attributes
C: PUBLIC CATALOGED
u: physical
C: SIZE - 12 TRACKS. LOCATED ON D2734. UNFORMATTED

In order to better understand the type of assistance applicable at each level of use, it is necessary to examine the requirements of users at each sophistication level.

4.1. PARROT

A Parrot operates in a simple "transcription mode." There is no consideration of input variability. The best form of input assistance is an example or a single choice from a single level menu system. The latter is analogous to function keys. By careful design, either of these approaches can be extended to assistance forms suitable for a Novice.

Only two basic responses can exist for the Parrot: the function completed successfully, or it was unsuccessful. If an unsuccessful response is provided, it can only state that the command was incorrectly entered and should be entered again (a Parrot does not comprehend the command's contents). If the system is unable to perform the task at this time, it can be suggested that the user try later. Since task completion is the reward for successful command entry, this information should always be provided to the user.

Thus at the Parrot level there is only one type of input assistance: an example.

4.2. NOVICE

The Novice may not distinguish between the first three stages of a transaction (Task, Semantic, and Syntactic Analysis). Thus, these stages should not be differentiated if the user's perspective is to be reflected in the interface. The system should lead the user from the determination of the subtask(s), through the isolation of the correct command and the determination of its semantic components, to the encoding of the information.

Once the user is ready to provide data for the command, a number of techniques can be applied. As stated earlier, continuity between the first three stages is important; the user should be unaware of any distinct phase of the transaction. Since the traditional command format may be inapplicable to the Novice, menus could be used for stages I and II followed by a mixture of menus, dialogs, and "fill-in-the-blanks" for stage III.

This expands the syntactic assistance to two levels:

**Assistance
Type**

**Sophistication
Level**

**Example
Simplest Form**

**Parrot
Novice**

Irrespective of the technique, the computer should take the initiative; the Novice may not know what information is required, or even if it is available. Thus, it is incumbent upon the assistance system to announce its existence. Information for clarification, however, should be provided only upon demand. To do so automatically, may unnecessarily confuse or annoy the user.

Responses, aside from providing information to the user, should indicate the successful completion of the command in a non-null form (something more than a prompt). A Novice, lacking confidence in the ability to control the system, may require this positive reinforcement.

2.1. INTERMEDIATE

Because the Intermediate is familiar with the system, the user, not the computer, should take the initiative. An individual at this sophistication level has the ability to decompose a task into its subtasks and determine an appropriate command (Stage I). Since the components of the system are known to exist, even if not understood, information should be factored into the following topics: command semantics, command syntax, and field or keyword semantics and syntax. Since individuals generally employ a subset of commands (Huckle 1980), assistance is still required for those used less commonly.

Assistance in the semantic and syntax analyses (Stages II and III) require additional information. As a user gains experience with a command, defaults are better understood, overridden, or modified. Thus, the scope of the command perceived by the user is extended. The semantic and syntactic expansion of commands requires that two new levels of assistance must be added:

1. The most common form of the command. This will occur when some commonly defaulted items are overridden.
2. The command is used in its full form. This occurs when no item is defaulted.

Thus, the number of levels are increased to four:

**Assistance
Type**

**Sophistication
Level**

**Example
Simplest Form
Common Form
Full Concrete Form**

**Parrot
Novice
Intermediate
Intermediate**

When the semantics and syntax of a command are not complicated, two or even one of the above forms may fulfill the information requirements.

Because the Intermediate operates in a terse mode, abbreviated forms of the command should be provided. This includes, not only contracted forms of the strings within the command (name, keywords, flags, etc.), but the items that can be defaulted and the values supplied.

The layered approach for responses should be available. As in case of information required for the input of a command, the user should be able to request specific information. The advantages (terseness and specificity) of requesting specific information is offset by the need for a query language.

2.2. ADVANCED

The needs of the Advanced user differ from the Intermediate in three ways.

1. The transaction stages considered prior to entering a command require a different emphasis because data and control structures are now a part of the user's command repertoire.
2. There is a need for assistance in the monitoring of an executing command since they are executed in a "batch environment".
3. A different type of response structure is needed since it must be interpreted directly by the command within the software without human intervention.

Within the first two stages, an increase in the type of information exists, reflecting the added control and data structures employed by the Advanced user. These new structures may be implemented within an existing command or via new commands. Assistance and instruction in the methods of building macros, procedures and programs are useful for the Advanced user. These new functional elements are reflected not only in Stages II and III, but their concepts must be included in Stage I.

Control and data structures are now used in the development of procedures. This places additional demands upon the response segment. Whereas in the lower sophistication level interfaces, the responses must be understood by a human, in a procedures, responses must be understood by the software.

The abstract nature of the command requires additional syntactic information. When a command has constructs that relate only to these structures, they must exist only in the information supplied to the Advanced user. Thus, in addition to the three assistance levels applicable to the Novice and Intermediate users, a fourth level,

containing the expanded language view must be included. The five levels of assistance are shown below:

Assistance Type	Sophistication Level
Example	Parrot
Simplest Form	Novice
Common Form	Intermediate
Full Concrete Form	Intermediate
Full Abstract Form	Advanced

3. CONCLUSION

On a theoretical basis, it is possible to factor software user assistance information into three independent categories:

1. verbosity
2. user sophistication
3. task segmentation

Although it is possible to prepare guidelines for the further classification of information within each category, only experimental investigations will validate these suppositions. At the present time, studies of specific topics are in progress.

4. REFERENCES

Clark, I.A., 1980, How to "Help" Help, IBM Report HF022, IBM United Kingdom Laboratories Ltd. (Hursley Park).

Huckle, B.A., 1980, Designing a Command Language for Inexperienced Users, Command Language Directions (D. Beech ed.), 199-212 (Amsterdam: North-Holland Publishing Company).

Reiles, N., 1979, The Design and Implementation of User-Oriented Systems. Madison WI, Univ. of Wisconsin. Ph.D. Thesis.

Schneider, M.L., Wexelblat, R.L., and Jende, M.S., 1980, Designing Control Languages From the User's Perspective, Command Language Directions (D. Beech ed.), 181-198 (Amsterdam: North-Holland Publishing Company).

Shneiderman, B., and Mayer, R., 1979, Syntactic/Semantic Interactions in Programmer Behaviour: A Model and Experimental Results, Journal of Computer and Information Sciences 7, 219-239.

SYSTEM MESSAGE GUIDELINES:

POSITIVE TONE, CONSTRUCTIVE, SPECIFIC, AND USER CENTERED

Ben Shneiderman
University of Maryland
Department of Computer Science
College Park, MD 20742
January 27, 1981

*** Draft paper prepared for Workshop on Human Factors in Interactive Systems, Georgia Institute of Technology, March 26-27, 1981, Atlanta, Georgia.

Prompts, explanations, error diagnostics, and warnings play a critical role in influencing user acceptance of software systems. Programming and command languages and application systems are appreciated not only for the functionality they offer but for the phrasing of system messages in a specific implementation. This is true for batch systems, but it is more important for interactive systems in which the impact of a message is immediate and more dramatic.

The wording of prompts, advisory messages, and system responses to commands may influence user perceptions, but the phrasing of diagnostic messages or warnings about improper conditions is critical. Since errors occur because of lack of knowledge, incorrect understanding or inadvertent slips, the user is likely to be confused, feel inadequate, and be anxious. Messages with an imperious tone, which condemn the user for an error, can heighten user anxiety, making it more difficult to correct the error and increasing the chances for further errors. Messages which are too generic, such as the ubiquitous "SYNTAX ERROR", obscure "FAC RJCT 004004400400", or mystical "OC7" offer little assistance to the novice user.

These concerns are especially important with respect to the novice user whose lack of knowledge and confidence amplify the stress related feedback which can lead to a sequence of failures. The discouraging effects of a bad experience in using a computer are not easily overcome by a few good experiences. In fact, I suspect that systems are remembered more for what happens when things go wrong than when things go right. Although these effects are most prominent with novice computer users, experienced users also suffer. Experts in one system or part of a system are still novices for many situations.

Awareness of the difficulties that novices encounter has prompted the development of student-oriented compilers for some languages, which emphasize good diagnostic messages and even limited error correction. The early DITRAN effort (Moulton and Muller, 1967) and CORC (Freeman, 1964) were followed by the WATFOR/WATFIV compilers (Cress, Dirksen and Graham, 1970) and the PL/C compiler (Conway and Wilcox, 1973). These efforts demonstrate what can be accomplished if the developers are sincere about their concern for ease of use. PL/C and WATFIV are widely used in academic environments not only because of their diagnostic messages but also because of their rapid compilation speeds. These systems demonstrate that although there may be a greater development cost for good diagnostics, the production costs can be kept low. Although I am not aware of any controlled experimental research which proves that students using these compilers learn faster, make fewer errors or have a more positive attitude toward computers, these hypotheses are shared by many people. Rigorous human factors studies would be useful in evaluating the improvement brought about by these systems and would be helpful in convincing skeptics about the importance of designing good system messages.

Producing a set of guidelines for writing system messages is not an easy task because of differences of opinion and the impossibility of being complete. In spite of these dangers, I feel that producing such guidelines could yield better systems. Input parsing strategies, message generation techniques, and message phrasing can be changed without affecting system functionality. Hopefully, more attention to system messages will lead to instrumentation of systems to capture data on error frequency distributions. Such data will enable system designers and maintainers to revise error handling procedures, improve documentation and training manuals, alter instructional materials, or even change the programming or command language syntax. Focusing increased attention on system messages should compel system developers to include the complete set of messages in user manuals. This high visibility will produce even more concern for the quality of these messages.

These comments are the result of experience and subjective evaluation. Controlled psychologically-oriented experimentation would be useful in verifying these conjectures.

BE SPECIFIC

Messages which are too general make it difficult for the user to know what has gone wrong. The simple minded and condemning messages such as "SYNTAX ERROR" or "ILLEGAL ENTRY", or "INVALID DATA" are frustrating because they do not provide enough information about what has gone wrong. Improved versions might be "Unmatched left parenthesis", "Legal commands are: Send, Read,

File, or Drop", or "Days must be in the range of 1 to 31."

Even in widely appreciated systems like WATFIV there is room for improvement. Messages such as "INVALID TYPE OF ARGUMENT IN REFERENCE TO A SUBPROGRAM" or "WRONG NUMBER OF ARGUMENTS IN A REFERENCE TO A SUBPROGRAM" might be improved if the name of the subprogram were included and the correct type or number of arguments were provided. The APL system which has so many nice human factors-oriented features comes out poorly when evaluated for system messages. The extremely brief "SIZE ERROR", "RANK ERROR", or "DOMAIN ERROR" comments are too cryptic for novices and fail to provide information about which variables are involved. On the plus side, the standardization (most systems use the APL360 messages) of messages does make it easier for users to move from one system to another. I have long felt that language standardization efforts should include standardization of at least the fundamental messages.

Execution time messages in programming languages should provide the user with specific information about where the problem arose, what variables are involved and what values were improper. When division by zero occurs some processors will terminate with a crude message such as "DOMAIN ERROR" in APL or "SIZE ERROR" in some COBOL compilers. PASCAL specifies "division by zero" but may not include the line number or variables that the PLUM compiler offers (Zelkowitz, 1976). Maintaining symbol table and line number information at execution time so that better messages can be generated is usually well worth the modest resource expenditure.

Systems which offer a code number for error messages are also annoying because the manual may not be available and consulting it is disruptive and time consuming. In most cases, system developers can no longer hide behind the claim that printing complete messages consumes too many system resources.

BE CONSTRUCTIVE

Rather than condemning the users for what they have done wrong, where possible tell them what they need to do to set things right. Nasty messages such as "DISASTROUS STRING OVERFLOW. JOB ABANDONED." (from a well-known compiler-compiler), "UNDEFINED LABELS", or "ILLEGAL STA. WRN." (both from a major manufacturer's FORTRAN compiler) can be replaced by more constructive phrases such as "String space consumed. Revise program to use shorter strings or expand string space.", "Define statement labels before use", or "RETURN I statement cannot be used in a FUNCTION subprogram".

It may be difficult for the compiler writer to write code which accurately determines what the user's intention was, so the advice to be constructive is often difficult to apply. I believe that error correcting compilers should be extremely conservative for the same reason. Automatic error correction has the danger that users will fail to learn proper syntax, and become dependent on the compiler making corrections for them. For interactive systems the user can be consulted before corrections are automatically applied.

BE USER-CENTERED

By user-centered I mean that the user controls the system rather than the system directs the user what to do. This is partially accomplished by avoiding the negative and condemning tone in messages and by being courteous to the user. If the system will take a long time to respond to a command then the user should be informed with a simple estimate of the time. Prompting messages should avoid the imperative forms such as "ENTER DATA" and focus on user control such as "READY FOR COMMAND" or simply "READY".

Brevity is a virtue, but the user should be allowed to control the kind of information provided. Possibly the standard system message should be less than a line, but by keying a "?" the user should be able to get a few lines of explanation. Two question marks might yield a set of examples and three question marks might produce explanations of the examples and a complete description. The CONFER teleconferencing system provides appealing assistance similar to this. The PLATO computer assisted instruction system offers a special HELP button and other options to provide explanations when the student needs assistance.

The designers of the Library of Congress' SCORPIO system (Woody et al., 1977) for bibliographic retrieval understood the importance of making the users feel that they are in control. In addition to using the properly subservient "READY FOR NEXT COMMAND" the designers avoid the use of the words "error" or "invalid" in the text of system messages. Blame is never assigned to the user but instead the system displays "SCORPIO COULD NOT INTERPRET THE FOURTH PART OF THE COMMAND CONTENTS, WHICH IS SUPPOSED TO BE A 4-CHARACTER OPTION CODE." The message then goes on to define the proper format and present an example of its use.

USE AN APPROPRIATE PHYSICAL FORMAT

Although professional programmers have learned to read upper case only text, most novices prefer and find it easier to read upper and lower case messages. Messages that begin with a lengthy and

mysterious code number only serve to remind the user that the designers were insensitive to the real needs of users. If code numbers are needed at all they might be enclosed in parentheses at the end of a message.

There is some disagreement about the placement of messages in program listing. One school of thought argues that the messages should be placed at the point in the program where the problem has arisen. The second opinion is that the messages clutter the listing and anyway it is easier for the compiler writer to place them all at the end. This is a good subject for experimental study, but I would vote for placing messages in the body of the listing assuming that a blank line is left above and below the message so as to minimize interference with reading the listing. Of course, certain messages must come at the end of the listing and execution time messages must appear in the output listing.

Some application systems ring a bell or sound a tone when an error has occurred. This can be useful if the error could be missed by the operator, but it is extremely embarrassing if other people are in the room and potentially annoying even if the operator is alone. The use of audio signals should be under the control of the operator.

The early high level language, MAD (Michigan Algorithmic Decoder) printed out a full page picture of Alfred E. Neuman if there were syntactic errors in the program. Novices enjoyed this playful approach, but after they had accumulated a drawer full of pictures, the portrait became an annoying embarrassment. Highlighting errors with rows of asterisks is a common but questionable approach. Designers must walk a narrow path between calling attention to a problem and avoiding embarrassment to the operator. Considering the wide range of experience and temperament in users, maybe the best solution is to offer the user a choice of alternatives - this coordinates with the user-centered principle.

2. EXPERIMENTAL RESULTS

2.1 COBOL Compiler Messages

A pilot study was run to explore the impact of improved messages on the ability of programmers to locate and repair bugs. The experiment, carried out by Patrick Peck and David Fuselier under the direction of the author, was administered to 22 second term COBOL students at the University of Maryland in Fall 1979.

Five bugs were included in a 132 line COBOL program yielding the

following messages from a UNIVAC COBOL compiler:

- 1) RESERVED WORD USED AS PARAGRAPH OR SECTION NAME IGNORE
ATTEMPT RECOVERY HERE AFTER PREVIOUS ERROR
- 2) DANGLING ELSE OR WHEN; TREATED AS AN IMPERATIVE
- 3) UNDEFINED DATA ITEM STATEMENT OMITTED
ATTEMPT RECOVERY HERE AFTER PREVIOUS ERROR
PREVIOUS ERRORS CAUSE LOSS OF OBJECT CODE
- 4) WORD NOT A VERB; SCAN SKIPS TO NEXT VERB
ATTEMPT RECOVERY HERE AFTER PREVIOUS ERROR
- 5) BLANK MISSING BEFORE OPERATOR OR LEFT PARENTHESIS
BLANK MISSING AFTER ARITH/COND OPERATOR OR PUNCTUATION

A second version of the listing was produced with the following five improved messages:

- 1) PERIOD IN PREVIOUS LINE CONTAINED IN IF STATEMENT, DELETE
- 2) EXTRANEIOUS ELSE IN PREVIOUS LINE, DELETE
- 3) BLANKS IS UNDEFINED DATA ITEM, MUST USE SPACES
- 4) USE AFTER PAGE INSTEAD OF AFTER 1 PAGE
- 5) SPACE REQUIRED BEFORE OPERATOR
SPACE REQUIRED AFTER OPERATOR

Code numbers and severity levels were eliminated in the improved messages and a single blank line was left above and below the improved messages. Eleven copies of each of the listings were produced and randomly distributed to the subjects. Seven minutes were allowed to locate and repair the bugs. One point was given for locating the error and two points were given for correcting the bug, yielding a maximum score of 10 points.

Subjects with the UNIVAC COBOL compiler listing had an average of 6.6 points while those with the improved messages had an average of 8.5 points. A t-test yielded a significant difference at the 5% level.

The results of this pilot study should be considered exploratory. Replications should be performed with other messages, professional subjects, and different languages. A more realistic study could be performed if two versions of the same language compiler were available. One group of subjects would be required to work with the standard version and the other group of subjects would work with the improved message version. Capturing performance in actual projects over longer time frames could

demonstrate the true impact of improved messages.

2.2 COBOL Compiler Messages: Tone and Specifity

2.3 Presence or Absence of Text Editor Messages

2.4 Tone and Content of Text Editor Messages"

2.5 Job Control Language Messages

3. CONCLUSIONS

REFERENCES

-- Extended Abstract --

Empirical Evaluation with Novice Users of Some
Programming Language Constructs

Elliot Soloway and Jeff Bonar

Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

This work was supported by the Army Research Institute for the Behavioral and Social Sciences, under ARI Grant No. MDA903-80-C-0508.

Any opinions, findings, conclusions or recommendations expressed in this report are those of the authors, and do not necessarily reflect the views of the U.S. Government.

I. Introduction

Language designers and language proponents are often given to making claims about the "readability," "debug-ability," "understandability," "learnability," "naturalness," etc. of a (their!) particular programming language. For the most part such claims are psychological in nature, and thus open to empirical inquiry. The problem is that this type of research is difficult to carry out and, frankly, only lip service (and "lip resources") to its need is given by the computing community. Moreover, with the major push behind Ada and methodologies appropriate to large scale software development, the needs of novice programmers have gotten particularly short shrift. We increasingly see the attitude that a "programmer" is a person who works on a 100 person team on some massive project -- not someone tailoring their home "mail network" or interacting with a computerized -- "programmable" -- toy. This view of programming seems a bit narrow.

With that introductory polemic, let us turn to the specifics of our presentation. We have been looking at how novice Pascal users cope with problem solving in Pascal. {1} In this extended abstract we shall first highlight several Pascal constructs which are particularly troublesome. Next, we shall make a more general statement, based also on empirical data, on the need to keep proceduralism in programming languages.

II. Performance Analysis: Read i/Process i vs. Process i/Read Next-i

Consider problem 3 in Table 1. For this problem, the stylistically correct solution in Pascal requires a curious coding structure:

```
read first-value
while (test ith value)
  process ith value
  read next-ith value
```

The loop must not be executed if the test variable has the specified value, and this value could turn up on the first read; thus, a read outside the loop is necessary in order to "get the thing going." However, this results in the loop processing being "behind the read; it processes the ith input and then fetches the next-i. We call this structure "process i/read next-i."

{1} One goal of our project, which will not be reported on in this summary, is to build a Run-Time Support Environment for novice Pascal users. This system, components of which are currently being built, will attempt to catch run-time bugs (not compile time errors, which are adequately handled in other systems) in students' programs, and provide remediation with respect to the underlying mental misconceptions.

Problem 1. Write a program which reads 10 integers and then prints out the av age. Remember, the average of a series of numbers is the sum of those numbers divided by how many numbers there are in the series.

Problem 2. Write a program which repeatedly reads in integers until their sum is greater than 100. After reaching 100, the program should print out the average of the integers entered.

Problem 3. Write a program which repeatedly reads in integers until it reads the integer 99999. After seeing 99999, it should print out the correct average. That is, it should not count the final 99999.

Table 1. Problems used in our test instrument. These problems were given to an introductory programming class on the last day of the course. They are designed to test student knowledge of key differences between different loop constructs in Pascal.

program Student6_Problem3;

```
var Count, Sum, Number : integer; Average : real;

begin
  Count := 0;
  Sum := 0;
  Read (Number);
  while Number <> 99999 do
    begin
      Sum := Sum + Number;
      Count := Count + 1;
      Read (Number);
    end;
  Average := Sum / Count;
  Writeln (Average)
end.
```

Figure 1 A stylistically correct solution to problem 3 in table 1. Note the need for two Read calls and the curious "process the last value, read the next value" semantics of the loop body. This program was minimally edited for presentation here. Students wrote these programs in a classroom. They were never submitted to a translator.

program Student7_Problem3;

```
var N, Sum, X : integer;
    Average : real;
    Stop : boolean;

begin
  Stop := false;
  N := 0;
  Sum := 0;
  while not Stop do
    begin
      Read (X);
      if X = 99999
        then Stop := true
        else begin
              Sum := Sum + X;
              N := N + 1;
            end
    end;
  Average := Sum / N;
  Writeln (Average)
end.
```

program Student16_Problem3;

```
var Count, Sum, Num : integer; Average : real;

begin
  Count := -1;
  Sum := 0;
  repeat
    Count := Count + 1;
    Read (Num);
    Sum := Sum + Num;
  until Num = 99999;
  Sum := Sum - 99999;
  Average := Sum / Count;
end.
```

Figure 2 These programs are attempts at problem 3 described in table 1. They are typical of the contortions students will go through to make this problem fall into a "read a value, process that value" frame. These programs have been minimally edited for presentation here. Students wrote these programs in a classroom. They were never submitted to a translator.

	<u>Read 1/Process 1</u>			<u>Process 1/Read Next-1</u>		<u>Other</u>
	<u>used</u>			<u>used</u>		
	<u>repeat loop</u>	<u>while loop</u>	<u>other</u>	<u>repeat loop</u>	<u>while loop</u>	
Correct	4	2			2	1
Incorrect	3	5	4			2

Table 2

The numbers in this table refer to the actual number of students, not percentage.

One of the authors -- the one with less Pascal experience -- intuitively felt this coding strategy to be unnecessarily awkward and downright confusing. Perhaps a more "natural" coding strategy would be to read the *i*th value and then process it; we call this the "read *i*/process *i*" coding strategy. Others have noticed this problem before, but treated it largely as a coding inconvenience. Their response was baroque looping constructs which eliminated writing the same code twice. We are not as concerned with elegance as with learnability. Do novice programmers use the stylistically correct coding strategy (process *i*/read next-*i*), or do they add extra machinery to a while or repeat loop (e.g., an embedded if test tied to a boolean variable) in order to force the code into a read *i*/process *i* structure?

Table 2 lists the performance of those students who attempted the problem with either a while or repeat loop. Of the 9 who solved it correctly, only 2 used the stylistically correct "process *i*/read next-*i*" coding strategy. (See Figure 1 for a solution using this coding strategy.) In order to correctly solve the problem using either a repeat or while loop and the read *i*/process *i* coding strategy requires extra machinery; Figure 2 shows student programs which use this strategy. Nonetheless, the vast majority of students attempted this solution; given the extra complexity needed for a correct solution, it is not surprising that many failed.

It is tempting to conclude that with respect to these types of problems, Pascal requires that students circumvent their "natural" problem solving intuitions. Before we can actually assert this conclusion, more research needs to be done {1}. But, since we must live with Pascal for some period of time to come, it would only be responsible for teachers to explicitly teach their students about this peculiar coding strategy.

{1} We have designed and pilot-tested the following experiment: we first ask all students to write a plan or design for problem 3 in Table 1 (the same one examined in this section), in a language other than a programming language. We then ask half the students to write the program in Pascal. For the other half of the group, we provide a one page description of constrained version of the Ada loop ... end loop construct in which only one exit from the loop body is allowed. While the sample size was small (13 students in total), the data is suggestive: invariably the plan of the students was worded in terms of a read *i*/process *i*. However, the Pascal versions were typically coded with a process *i*/read next-*i* strategy. But, those programs written using the Ada loop ... end loop were coded using the read *i*/process *i* strategy. Thus, the program coded in Ada more closely matched the students' plans than did those program coded in Pascal. We plan to run this experiment on a larger group.

III. Performance Analysis: Getting a New Value

In all 3 problems (Table 1), a correct solution required that the program "get a new value with a read." 23% of all the student written programs did not perform this function correctly. Often students try to get the previous or next value of a variable by subtracting or adding one (see Figure 3). {1} We also found programs in which we felt students assumed that each use of Next_value automatically retrieved a new value.

As "expert programmers" we have a great deal of deep knowledge about how to program. In particular, we know that variables have not just types, but also roles. Different coding strategies are needed to realize like operations on variables whose roles are different. For example, "getting the next value" implies adding one for a counter variable, reading for a New_value variable, and adding in the New_value for a Running_total variable. (The problems in Table 1 need one variable in each of these roles.) Perhaps students committing the above errors did not understand or garbled these different variable roles.

Misunderstanding this "deep" knowledge about Pascal -- mind bugs -- could result in many different student errors -- surface bugs. Perhaps students committing the above errors did not understand that read is actually just a special case of assignment. If so, then a language which treated I/O calls as special values which can be assigned "to" or "from" might be more palatable to beginning programmers, e.g.,

```
New_value := Read_from_terminal, or,  
Write_to_terminal := Running_sum / Count.
```

Another possible mind bug which could result in some of the observed errors would be that students incorrectly overgeneralized from the Counter variable. That is, since the next value of a variable functioning as a counter can be retrieved by simply adding a 1 to the variable, why not get the next value of any variable by simply adding a 1 to it! While reasonable, this is incorrect.

IV. Performance Analysis: The "Demon" in the while loop test

Based on our examination of student programs, and on analysis of audio-taped, individual interviews, we felt that there was a great deal of confusion surrounding the time at which the terminating test in the while loop gets evaluated: is it

{1} "Backing up" may be needed when a student does problem 3 in table 1 with a read i/process i strategy.

program Student30_Problem2;

```
var N, Sum, Score : integer; Mean : real;
begin
  N := 0;
  Sum := 0;
  Score := 0;
  while (Sum <= 100) do
    begin
      Score := Score + 1;
      Sum := Sum + Score;
      N := N + 1;
    end;
  Mean := Sum / N;
  writeln ('the mean = ', Mean:10:10)
end.
```

program Student19_Problem1;

```
var Num, Prev_num, Count : integer;
begin
  Count := 0;
  Read (Num);
  Sum := 0;
  repeat
    Prev_num := Num - 1;
    Sum := Num + Prev_num;
    Sum := Sum + 1;
    Count := Count + 1;
  until Count = 10;
  Average := Sum / Count;
  WriteLn ('Average of ten integers is equal to ',2)
end.
```

Figure 3 These programs are attempts at the problems described in table 1. They illustrate student problems with getting a New value. These programs have been minimally edited for presentation here. Students wrote these programs in a classroom. They were never submitted to a translator.

Problem 1:

Given the following statement:

"At the last company cocktail party, for every 6 people who drank hard liquour, there were 11 people who drank beer."

Write a computer program in BASIC which will output the number of beer drinkers when supplied (via user input at the terminal) with the number of hard liquour drinkers. Use H for the number of people who drank hard liquour, and B for the number of people who drank beer.

Sample Size	% Correct	% Incorrect
52	69	31

Problem 2:

Given the following statement:

"At the last company cocktail party, for every 6 people who drank hard liquour, there were 11 people who drank beer."

Write an equation which represents the above statement. Use H for the number of people who drank hard liquour, and B for the number of people who drank beer.

Sample Size	% Correct	% Incorrect
51	45	55

Probability of these results on the assumption that errors on each problem were equally likely is $p < .05$

Table 3

evaluated once, at the top of the loop, or is the test continually evaluated during the execution of the body of the loop? The program given below was also on a written test taken by the 31 summer school students.

Program Problem4:

```
var Count : integer;  
  
begin  
  Count := 0;  
  while Count < 7 do  
    begin  
      Writeln ('*');  
      Count := Count + 1;  
      Writeln ('/')  
    end  
  end.
```

If the students felt that the terminating test was evaluated continually, then the loop should terminate before an '/' were printed, thus providing one more '*' and '/'.^{1} In other words, it is as if the test were a "demon" watching the statements in the loop body, and waiting for its condition to become true. Of the 31 students, 34% made the above mistake. Given the ubiquity of the while construct in programs and in the instruction, and given the lateness in the course (the end of the semester), we felt that this was a surprisingly high percentage.

We feel that the basis for this confusion is grounded in the mismatch between the semantics of while in a programming language context, and the semantics -- the meaning -- of 'while' in "every day experience." In the latter case, 'while' has a global sense: during the course of some event. In contrast, the programming language while requires a local, narrow interpretation: at a specific point in time. Clearly, the names of programming language constructs must rely on real world semantics of their analogs. However, care ought to be exercised in their selection. Again, we are unlikely to change Pascal or the while loop construct, but educators must take note of this error, and pay attention to it in their instruction.

V. The Need for Proceduralitu in Languages for Novices

^{1} We were not interested in the actual number of '*' and '/', i.e., we were not studying the "off-by-one" bug in this particular problem.

There is a definite trend in programming language design and programming methodology towards more "formality." For example, "logic" and production rules have been seriously suggested as programming languages. Dijkstra suggests that the process of writing a program should be akin to that of writing a mathematical proof. Backus' new language takes a different, yet similar approach: take procedurality out of the programming language and make it algebra based to facilitate program proofs. While these languages and approaches might be appropriate for experts, we are quite skeptical of their appropriateness for novices. We are seriously concerned that programming not be equated with mathematics. For whatever reasons, most people have a great deal of trouble learning and using mathematics. We believe, and we are not alone, that there are aspects of programming which uniquely lend themselves to the demystification of mathematics. The formal programming people propose to remove exactly those aspects of programming while increasing required math ability. In our increasingly sophisticated world, just plain folks will need to "program", and our formal programming friends have no answers for these non-professional programmers. We are not willing to write off just plain folks.

In the following, we take a less polemical, and more evidence based look at one of the "unique aspects of programming" alluded to above, namely, procedurality.

Procedural vs. Non-Procedural: That is the Question

The first study which we feel supports the need to keep procedurality in programming languages for novices was done by Welty and Stemple [1981]. They compared the ability of novice subjects to write database queries in languages with different amounts of procedurality. In all issues except procedurality, the languages were identical. A typical query in SQL, the less procedural language, is:

```
SELECT NAME
FROM STUDENTTABLE
WHERE HOMESTATE = 'OHIO'
```

The equivalent query in TABLET, the more procedural language, is:

```
FORM OHIOANS FROM NAME, HOMESTATE OF STUDENTTABLE
KEEP ROWS WHERE HOMESTATE = 'OHIO'
PRINT NAME
```

In their paper they formalize "amount of procedurality" based on the number of variables, the number of operations, and the degree to which the bindings and operations are ordered by the language semantics. The two languages were learned by subjects working largely on their own. The same example problems and order of presentation was used for each group. The experiment showed that subjects who learned the more procedural query language, TABLET,

wrote difficult queries better than those using the less procedural language SQL.

The second study which we feel supports our claim is being carried out by Soloway and his colleagues at UMASS. In our work, we explored the performance of students on "ratio" type word problems. Typically, half the students in a low-level programming class were asked to solve a word problem with an algebraic equation, while the other half were asked to solve the same problem with a program (Table 3). As the results indicate, significantly more students got the problem correct in the programming context than did those in the algebraic context. A number of these experiments have been run in which various parameters were varied (e.g., problem wording). In all cases the results were similar to those in Table 3.

We have a number of specific hypotheses which could account for this performance difference. The basis for all of them, however, is procedurality. Some students who used algebra as the solution language seemed to view the equation as a "picture description:" there are more beer drinkers than hard liquor drinkers, thus $11B$, which represents the beer drinkers, is related to $6H$, the hard liquor drinkers, via $11B = 6H$. Alternatively, some students viewed the algebraic equation as "label descriptors," much like "3ft. = 1yd." (1) On the other hand, programming appears to encourage students to view the equation as an active operation, or transformation. That is, the fact that variables have values, and that variables are acted upon by operations, appear more understandable to students in the programming environment. Thus, the procedural nature of programming seems to be a key factor in understanding and using such basic concepts as variable, operation, equal sign.

Concluding Remarks

Clearly, this note is only a "teaser," a fuller discussion of these issues must await the workshop. We genuinely solicit your comments, and look forward to an active interchange at the workshop.

(1) These hypotheses are based on the analysis of many hours of video-taped clinical interviews with individual students as they solved problems of the above sort.

**Steamer: An Advanced Computer Aided Instruction
System For Teaching Propulsion Engineering**

Albert L. Stevens

Michael D. Williams

James D. Hollan

In this presentation, we describe the current state of Steamer, an intelligent CAI system with a graphics-based human interface. Steamer includes a math model of a steam plant, an interactive graphics front end and a qualitative modelling component. The math model and graphics interface allows the student to control and observe a simulated steam plant. The qualitative modelling component enables Steamer to explain in casual terms the operation of components and subsystems. The design of the graphics interface is based on object-oriented programming to allow much more modularity and flexibility than is normal with computer graphics. The qualitative modelling component is based on incremental qualitative simulation to model systems in terms of psychologically meaningful events.

METAMORPHOSIS THROUGH METAPHOR

J.C. THOMAS

IBM CHQ Armonk, NY

The problems that mankind faces in the twentieth century sometimes seem insurmountable. Nuclear weapons, the population explosion, rising demand and falling levels of most natural resources provide a potentially devastating combination. In addition, our new lifestyles have provided a number of unwelcome ecological surprises.

The organism and the environment are necessarily in an intimate relationship. Yet, we humans are, seemingly by choice, changing our environment much faster than we can adapt biologically. It seems suicidal.

The only major way out of these dilemmas is for effective human intelligence to increase dramatically over the next century. This could theoretically be accomplished biochemically, educationally, or through more effective group problem solving procedures.

The fourth possibility, which is addressed in this paper, is that of the computer augmenting effective human intelligence. By augmenting effective human intelligence I mean that by using a computer, people will operate so as to bring greater short and long term happiness to themselves, to mankind, and to life than they will without the computer.

The major obstacle to this goal is not the lack of progress in computer technology: we are able to build smaller, faster, cheaper components. (That progress, of course, is what enables us to address the next problem). What we have been slow to achieve is a computer that is anything near optimally designed to help a human being do a more effective, higher quality job. In order to accomplish this latter goal, we need some notion of what humans can do, what they need to be able to do better in order to solve their problems and what the capabilities of the computer are.

In this paper, I will focus on part of this problem. First, I will present a model of how the person approaches and learns to use a new tool. Second, I will point out where in this process there is likely to be a critical breakdown which prevents the person from using the tool in an effective fashion (e.g., to solve previously insoluble problems). Third, I will present a theory of what the tool should look like and provide some suggestively supporting evidence based on experimental work of my own and of other investigators. Fourth, in the area of office systems, I will present some examples of how my recommendations might be implemented.

The model of mind is multi-viewed; at the current state of integration of behavioral science no single view (e.g., behavioristic or cognitive) provides as sufficient a scope as does a multi-viewed approach.

The presented model is novel in the context of human-computer interaction in the notion of resource allocations with differentially usable resources, in an emphasis upon motivational issues, and in the analysis of primary, secondary and tertiary memory limitations.

The model implies that under certain conditions a kind of "gambler's ruin" phenomenon will occur in which the aspiring learner of a potentially useful system will stop short. An even more common case of essentially the same phenomenon will occur among those learners who learn enough about the system to do what they did before only marginally better. Rarely, a user will learn an interface so that they are truly facile with the facilities.

Still rarer are cases in which the computer-tool allows a qualitative change in the user's work. Yet for augmenting effective human intelligence, it is this last category that we would like to contain the majority of users. For such a qualitative change to occur, the interface must be designed to allow a more optimal allocation of the user's psychological resources.

One way of accomplishing this latter goal is through the use of an appropriate metaphorical interface presented to the user along with a well thought-out mapping inside the computer system that translates the actions the user takes in the metaphorical space into the appropriate state changes in the machine, and translates the machine state changes into the appropriate presentations in the user's metaphor.

A large body of empirical evidence strongly suggests that "meaningful" material can greatly affect the user's performance quantitatively and in some cases qualitatively. "Meaningfulness" can exist at many levels. Editing commands that are more English-like are better than their abbreviational counterparts (Ledgard, et als (1980). Non-programmers can learn an English-like query language better than its symbolic counterpart (Reisner, 1975). Older subjects particularly, but younger ones as well, are aided in learning by the addition of "extra" mnemonic material (Thomas & Rubin, 1972).

The implications of these findings for a particular domain - office systems is drawn in some detail. A number of objects, organizing schemes, features, and actions that

people are familiar with are reviewed along with the way in which these can be combined to let the user know what is going on. The model explains how using such metaphors can increase comprehension, motivation, and performance of given tasks and how such metaphors can be used to improve the effective intelligence that goes into the user's solutions.

In addition to using metaphors, a better allocation of the user's psychological resources can be achieved by making more complete use of various input and output characteristics of human beings. People can discriminate better when information is presented on a large number of channels (rather than a single channel). People can also output at greater data rates over several channels. In traditional, pencil and paper editing, non-verbal, spatial symbols are used as the metalanguage for the verbal material. In film directing, on the other hand, much of the metalanguage is verbal. We need to become more sensitive to this kind of "division of labor" in our computer interfaces.

A SYSTEM FOR COMPUTER AIDED MEMORIZATION

Michael D. Williams

Xerox Palo Alto Research Center

Palo Alto, California

James D. Hollan

Navy Personnel Research and Development Center

San Diego, California

and

University of California, San Diego

La Jolla, California

We are constructing an intelligent computer based instructional system to facilitate students in the memorization of a large collection of facts. The system consists of a series of games played on a microprocessor, a relational data base to drive the games, a student model, and a computer coach. To the student the system appears as a series of games played with a table top computer against a computerized opponent. Example games are twenty questions, flash cards, a property specification game where students successively enhance the definition of an object until one or no objects match the cumulative description, a picture recognition game, and a concentration-like table fill-in game. The data base can be modified to allow a variety of topic matters. Present data bases include US and Russian ships, their radars and weapons, South American geography, the anatomy of the human hand, and a fantasy data base on star trek trivia. The student model consists of a simple marking of the relations in the data base. The computer coach consists of a series of opponents of variable "intelligence" and a scheme for focusing game activity on portions of the data base where the student is weak and the information important.

Our principle student population are Naval Officers learning the properties of Russian ships, radars, and weapons. The data base they are attempting to master consists of thousands of facts. Approximately 3 and 1/2 weeks of a 6 week course on tactical decision making are taken up with lectures, practice, and tests to support this memorization.

Our primary scientific goal in this work is to explore the process of remembering. We are using this computerized memorization system as a tool to gather data as well as a forcing function to drive the development of our theory. An issue that anyone building a computerized

instructional system must confront is *what* information to present a student and *when* to present the information. The goal for our theory of remembering is to determine the implications of learning any particular piece of information with regard to the durability of what the student knows, flexibility of retrieval, errors in recall, incidental information recovered, and speed of retrieval.

We come to the problem with the view that remembering is a complex process of reconstruction from an array of fragments. An essential observation is that people memorize *more* than just the facts in the data base. A large amount of their learning seems to focus around abstractions drawn, in part, from the regularities within the data base. Thus, a student might notice that all ships which carry a scoop-pair radar also carry shaddock missiles (this is because the scoop-pair radar is the guidance radar used to control that particular missile, it has no function without the missile). In effect, students seem to be building a "theory" of the data base from which they can reconstruct the portion they need to answer any given query. Given that this is the case, what we are looking for are the particular mnemonic effects of these "abstractions", and principled reasons for these effects within a reconstructive theory of remembering.

Our primary engineering goal in this work is to build a system which provides substantial facilitation to students who must memorize some collection of facts. In this role we are investing substantial efforts in what we call the *pragmatics* of the system design. Thus we are using computer games to enhance motivation, have spent large amounts of time designing and tuning the interface between student and machine, and are using a technique of *in situ* development to tune the system toward realistic user needs.

Names and Addresses of Participants

Albert N. Badre	School of Information and Computer Science Georgia Institute of Technology Atlanta, Georgia 30332 (404)-894-2598
Richard Burton	Xerox PARC 3333 Coyote Hill Road Palo Alto, California 94304 (415)-494-4000
Jaime G. Carbonell	Carnegie Mellon University Department of Computer Science Schenley Park Pittsburg, Pennsylvania 15213 (412)-578-3064
Susan T. Dumais	Bell Laboratories 600 Mountain Avenue Murray Hill, New Jersey 07974 (201)-582-2054
Sam L. Ehrenreich	U.S. Army Research Institute Attention: Peri-OS (S.L. Ehrenreich) 5001 Eisenhower Avenue Alexandria, Virginia 22333 (202)-274-8905
Jim Foley	Department of Electrical Engineering and Computer Science George Washington University Washington, D.C. 20052 (202)-676-4952
George W. Furnas	Bell Laboratories 600 Mountain Avenue Murray Hill, New Jersey 07974 (201)-582-6128
Stanley M. Halpin	U.S. Army Research Institute Attention: Peri-OS (Stanley M. Halpin) 5001 Eisenhower Avenue Alexandria, Virginia 22333 (202)-274-8905

Mark D. Jackson	Bell Laboratories Room 6A304B Warrenville and Naperville Roads Naperville, Illinois 60566
Janet Kolodner	School of Information and Computer Science Georgia Institute of Technology Atlanta, Georgia 30332 (404)-894-3285
Thomas K. Landauer	Bell Laboratories 600 Mountain Avenue Murray Hill, New Jersey 07974 (201)-582-4324
Michael Lebowitz	Department of Computer Science 406 Mudd Building Columbia University New York, New York 10027 (212)-280-2564
Paul R. Michaelis	Texas Instruments Computer Science Lab Post Office Box 225936 Mail Station 371 Dallas, Texas 75265 (214)-995-7081
Mark Miller	Texas Instruments Computer Science Lab Post Office Box 225936 Mail Station 371 Dallas, Texas 75265 (214)-995-7081
Franklin L. Moses	U.S. Army Research Institute Attention: Peri-OS (F.L. Moses) 5001 Eisenhower Avenue Alexandria, Virginia 22333 (202)-274-8905
Jean Nichols	U.S. Army Research Institute Attention: Peri-OS (J. Nichols) 5001 Eisenhower Avenue Alexandria, Virginia 22333 (202)-274-8905
Phyllis Reisner	IBM Department K54/282 5600 Cottle Road San Jose, California 95193

Elaine Rich	Department of Computer Science University of Texas Austin, Texas 78712 (512)-471-7316
Michael L. Schneider	Sperry Univac Post Office Box 500 Blue Bell, Pennsylvania 19424 (215)-542-4011
Ben Shneiderman	Department of Computer Science University of Maryland College Park, Maryland 20742 (301)-454-4245
Elliot Soloway	Department of Computer Science University of Massachusetts - Amherst Amherst, Massachusetts 01002 (413)-545-1324
Albert L. Stevens	Bolt Beranek and Newman, Inc. 50 Moulton Street Cambridge, Massachusetts 02238 (617)-491-1850
John C. Thomas	IBM Old Orchard Road Armonk, New York 10504 (914)-765-1900
Judith Tschirgi	Bell Laboratories Room 6A304B Warrenville and Naperville Roads Naperville, Illinois 60566 (312)-462-5976
Michael D. Williams	Xerox PARC 3333 Coyote Hill Road Palo Alto, California 94304 (415)-494-4000